What is the Connection Between Issues, Bugs, and Enhancements?

Lessons Learned from 800+ Software Projects

Rahul Krishna, Amritanshu Agrawal, Akond Rahman, Alexander Sobran*, and Tim Menzies
North Carolina State University, *IBM Corp
i.m.ralk@gmail.com,[aagrawa8,aarahman]@ncsu.edu,asobran@us.ibm.com,tim@menzies.us

ABSTRACT

Agile teams juggle multiple tasks so professionals are often assigned to multiple projects, especially in service organizations that monitor and maintain large suites of software for a large user base. If we could predict changes in project conditions change, then managers could better adjust the staff allocated to those projects.

This paper builds such a predictor using data from 832 open source and proprietary projects. Using a time series analysis of the last 4 months of *issues*, we can forecast how many *bug reports* and *enhancement requests* will be generated the next month.

The forecasts made in this way only require a frequency count of these issue reports (and do not require an historical record of bugs found in the project). That is, this kind of predictive model is very easy to deploy within a project. We hence strongly recommend this method for forecasting future issues, enhancements, and bugs in a project.

CCS CONCEPTS

• **Software and its engineering → Agile software development**;

KEYWORDS

Time series analysis, Bugs, Collaborations, Issues

ACM Reference Format:
Rahul Krishna, Amritanshu Agrawal, Akond Rahman, Alexander Sobran*, and Tim Menzies. 2018. What is the Connection Between Issues, Bugs, and Enhancements?. In *Proceedings of 40th International Conference on Software Engineering: Software Engineering in Practice Track, Gothenburg, Sweden, May 27-June 3, 2018 (ICSE-SEIP '18),* 10 pages.
https://doi.org/10.1145/3183519.3183548

1 INTRODUCTION

In the early days of software engineering, when doing any single project was a Herculean task, developers were often assigned to one project for months at a time. In the age of agile [1, 6, 14, 39], that has changed. How should management practices change to better accommodate agile developments?

One principle of agile is to use multitasking to get work done. Agile teams juggle multiple tasks, so professionals are often assigned to multiple projects, especially in service organizations that monitor and maintain a large suite of software for a large user base.

When project conditions change, it would be very useful if managers can adjust the staff allocated to those projects. Ideally, managers overseeing multiple projects would like to tell whether more/less developers will be required in the upcoming month in order to make informed choices about changes to staffing allocations.

This ideal scenario might be achievable, given access to enough software projects. For example, here we apply a time series analysis to 832 proprietary and open source projects from GitHub repositories. Data was gathered by sampling projects every week for an average period of two years. The mined data contained the following attributes: issue reports, bugs reports, and enhancement requests for each week. Temporal trends in this data were modeled using *AutoRegressive Integrated Moving Average* (ARIMA) [8] (this is a standard method that has shown to outperform other models such as linear regression or random walk [2, 9, 22, 29, 42]) [1].

Using this data, we offer the following contributions:

(1) In hundreds of software projects, we prove the existence of predictable trends in issues, bugs, and enhancements.
(2) These different trends are closely associated. Hence, using past issue reports we can forecast the future existence of (a) bugs and (b) enhancement requests.
(3) The forecasts made in this way only require a frequency count of these issue reports over time. They do not require an detailed nature of the bugs found in the project. Therefore, this kind of predictive model can be very easy to deploy within a project.
(4) In studies with 832 projects, we show that the forecasts made using issues are remarkably accurate. We show that forecasting using issues is statistically similar (with $p < 0.05$) to forecasting with historical logs of bugs and enhancements.

The rest of this paper is structured as follows. After some notes on the motivation of this work, this paper's research questions are presented in §1.1. §2 discusses some related work. §3 presents our experimental methodology. In §4, we answer our research questions and discuss the lessons learned. In §6 we discuss the practical implications of our results. In §5, we present our threats to validity. Finally, §7 presents conclusions and directions for future work.

ICSE-SEIP '18, May 27-June 3, 2018, Gothenburg, Sweden

ACM ISBN 978-1-4503-5659-6/18/05…$15.00
https://doi.org/10.1145/3183519.3183548

[1] Note that that this paper focuses on "near-term forecasts". Long term forecasts are beyond the scope of this paper. We leave that for future work.

1.1 Motivation and Research Questions

Why build yet another bug predictor? Why learn predictors for future bug reports using data from prior issues? In the literature there are any number bug prediction methods as witnessed by the plethora of papers in that area [31, 32, 37, 38, 48].

For large cloud-based environments of service organizations supporting multiple languages and tools, we found that those methods had certain significant drawbacks. Firstly, before anyone can use past bugs to forecast for future bugs, *they need access to past bugs*. For this, they could use APIs provided by platforms like GitHub to obtain temporal logs issues. However, this is not sufficient. After mining issues, these need to be carefully curated to identify the bugs. We spent over two months at IBM to manually categorize over 100000 logs of issues into bugs and enhancements. Due to the significant amount of time and effort required to do this, in this paper we explore what can be achieved *with just* logs of issues.

Secondly, there was the problem of commissioning standard defect predictors for dozens of programming languages[2]. Nearly all the prior defect prediction work in software engineering have placed focus only on a few languages like C++ and Java. Certainly, we could build our own but predictors, but merely building them is not the issue. Far more problematic is the issue of certifying that they work against a known baseline data (which may be missing). In addition to that, maintaining all that software over all those languages would turn into an arduous task in itself.

In meetings to discuss alternatives to (a) using traditional defect prediction technology, and (b) having to manually categorize issues as bugs and enhancement, we came across a simple alternative—forecast bugs by looking at trends of past issue reports. We found that by *mining only for issues*, we can construct accurate models that forecast for bugs and enhancements. To show that, this paper explores the following research questions.

RQ1: Are there temporal trends in our data?

Motivation: The first research question seeks to establish the existence of temporal trends in the attribute we have mined (i.e., issues, bugs, and enhancements). To assert this, we ask if the past temporal data of attributes mined can be used to construct time series models that forecast future trends for the same attributes.

Approach: For each of our 832 proprietary and opensource projects, we use the mined attributes (issues, bugs, and enhancements) and for each of these attributes we construct a time series model with ARIMA. Then, we used these ARIMA models to forecast future issues, bugs, and enhancements. (Note: other than in RQ1, we built ARIMA models only on issues and used those to forecast for bugs and enhancements.)

> ***Result:*** ARIMA models built on past temporal data of issues, bugs, and enhancements can be very accurate for forecasting future values.

RQ2: Are there correlations between mined attributes?

Motivation: Our second research question follows the report by Ayari et al. [3] regarding the correlation of issues reports with bugs and enhancement. Here we seek to establish this on our dataset of 832 projects. We ask if the time series trends of issues, bugs, and enhancements are correlated to each other. A strong correlation between these attributes would enable use to make use of models built on one attribute such as issues to forecast for other attributes.

Approach: In each of our 832 proprietary and opensource projects, we compute the Spearman's ρ value between pairs of attributes. A value close to 1 would indicate a strong positive correlation, a value close to -1 would indicate a strong negative correlation, and a value close to 0 would indicate no correlation.

> ***Result:*** In proprietary projects certain pairs of attributes such as $\langle issues, bugs \rangle$ and $\langle issues, enhancements \rangle$ exhibit a reasonable correlation. In opensource projects, on the other hand, the correlations between project attributes still exist but they are relatively weaker in comparison to proprietary projects.

RQ3: Can issue reports forecast for future bugs and enhancements?

Motivation: This research question naturally follows RQ2. Here we ask if it is possible to use time series models built on one attribute such as issues to estimate for another attributes such as bugs and enhancements.

Approach: We construct an ARIMA model on time series data of issue reports for each project to forecast for bugs and enhancements. That is, we transfer ARIMA models between:

a) issues → bugs, and
b) issues → enhancements.

Then we compare the forecast values with the actual values by measuring the magnitude of residual error.

> ***Result:*** ARIMA models built on issues can be very accurate for forecasting future bugs and enhancements.

RQ4: Are the forecasts using issues better than with using past temporal data?

Motivation: In the final research question, we compare the errors between using time series model built with issues to forecast for future values of bugs and enhancements with forecasts using manually labeled past temporal data of bugs and enhancements. As mentioned previously, it took us a significant amount of time to curate issue reports into bugs and enhancements. If the errors in using only issues for forecast is statistically comparable to using each time series separately, then we can establish that the time series trend of issues can indeed forecast for bugs and enhancements and that may save a lot of time and effort.

Approach: As before, for each of our 832 proprietary and opensource projects, we construct two time series models with ARIMA:

a) *ISSUE* : ARIMA model built using past issue report trends.
b) *LABELED* : ARIMA model built using manually labeled historical bug reports and enhancement requests.

We use both of these models to forecast for future bugs and enhancements. Then, we compute the error in forecasts with *ISSUE* and *LABELED*. Finally, we use a statistical test (Welch's t-test) to compare the errors.

> ***Result:*** Forecast errors of *ISSUE* are statistically similar to *LABELED*. That is, we can avoid all the complexity of bug mining by just building times series models from issue reports.

[2]Popular languages include Java, Python, Javascript, C++, Lua, Perl, Ruby, etc.

2 RELATED WORK

The study of evolution of software systems over time has been subject to much research over the past decades. Several researchers have attempted to model long term temporal behavior of aspects of software systems such as structural changes, lines of code, etc. [18, 33]. Godfrey and Qiang studied the growth of a number of open-source systems such as linux and gcc compilers. For linux, they report that the growth rate is geometric. Using this, they were able to develop a time series approach to model long term growth of such systems. Such time series models have been very popular with several software engineering researchers [16, 33, 49, 55]. Fuentetaja and Bagert [16] used time series growth models of software systems to forecast how much memory systems may use. They demonstrate that these time series growth models of software systems exhibit a power law. Using a Detrended Fluctuation Analysis method they were able to establish a theoretical basis for some trends they noticed in software evolution. Wu et al. [55] studied the existence of correlations in time series over long stretches of time. Using a Re-scaled Range Analysis technique [23], they reported the existence of temporal signature in the software systems and that these systems exhibit a macroscopic behaviour of self-organized criticality.

Another area of software engineering that has seen extensive use of time series modeling is software reliability. Several researchers, from as early as 1972 [24], have attempted to describe time series models for measuring software reliability. This has resulted in several 100 different forms of time series models [35]. These initial models made strong stochastic assumptions about the data and were grouped in to two categories: (1) Failure interval models [19, 24], and (2) Failure count models [20, 36]. However, these models made several unrealistic assumptions such as independence of time between failures, immediate correction of detected faults, and correcting faults without introducing new faults [19, 54, 59]. Zeitler et al. [59] cautioned that this was a major impediment since, the real world use of these models was not practical.

In response to the above, researchers explored approaches based on non-parametric statistics [5, 44] and Bayesian networks as possible solutions [4, 15, 41, 53]. However, even though these non-parametric approaches are able to address the unrealistic assumptions issue; they cannot completely address the applicability and predictability issues. As a result, other methods based on neural networks and other machine learning methods [12, 27, 30, 34, 43, 57, 58] were introduced. However, the issue with these approaches is that they require a large training data set as input/output examples to obtain accurate forecasts, which was computationally intensive and time consuming process.

An alternative to the above approaches was offered by Zeitler et al. [59]; they recommended the use of time series models such as ARIMA. They offer strong statistical evidence to show the time series models (especially ARIMA) are best suited for mapping system failures over time [59]. As a result, a number of researchers have applied time series models, especially ARIMA models [2, 10, 25, 46, 56]. These researchers have shown that ARIMA models have the ability to give accurate forecasts [2]. Yuen et al. [11] used ARIMA models to predict the evolution in the maintenance phase of a software project with sampling periods of one month. Kemerer et al. [28] use ARIMA models to predict the monthly number of changes of a software project. Herraiz et al. [21] used ARIMA models to model time series changes in Eclipse IDE by smoothing using kernel methods.

Although ARIMA models have been well established for time series analysis, our reading of the literature has indicated that many of these methods do not perform a comprehensive empirical study to establish it's usefulness. The success of ARIMA models from Ayman et al. [2] for instance was shown to work on 16 projects (some as small as 5000 LOC). Similarly, Kenmei et al. [29] et al. performed their analytics on only 3 software systems, Eclipse, Mozilla, and JBoss. Our paper overcomes this limitation by performing a large-scale case study with the ARIMA model on 832 open source and proprietary applications. In doing so, we demonstrate promising results showing that it is possible to mine projects on GitHub over long stretches of time to generate time series models which in turn can be used to forecast the number of issues, bugs, and enhancements.

3 METHODOLOGY

This section first provides a formal definition of issues, bugs, and enhancements in §3.1. Then we detail our datasets and our policy for gathering and filtering these datasets in §3.2. Then, we discuss time series modeling with ARIMA in §3.3. After that, we discuss the proposed forecasting approach and statistical measures in §3.4 and §3.5 respectively.

3.1 Terminologies

In our work, we mine repositories for *issues, bugs*, and *enhancements*. We used the following definition for these attributes.

- *Issues:* GitHub offers issues as an in-built way to keep track of tasks associated to projects. They can be shared and discussed with the rest of the team. Each issue may be associated with a color-coded *tags* to help categorize and filter them.
- *Bugs:* Github issues are most widely used to track bugs and defects in the project. This is analogous to other bug tracking systems such as bugzilla or JIRA. In GitHub, bugs can be identified as issues with a tag "bug" assigned to them.
- *Enhancements:* In additional to bug tracking, users often use enhancements to make specific requests e.g, for new features. These requests are usually tagged as "enhancements".

Note that, while issues for a project can be easily mined using GitHub's API the same is not true for bugs and enhancements. These had to manually extracted due to GitHub's API call limits. This manual extraction and coding process can be quite laborious.

3.2 Datasets

To perform our experiments we use open-source projects from GitHub, and proprietary projects obtained from our industrial partners at IBM Raleigh. These totaled to 1,646 different projects with 1,108 opensource projects and 538 proprietary projects. After filtering (see §3.2.1), these were reduced to 832 projects. Our data selection strategy is as follows:

(1) In case of open source projects we select public GitHub projects that are included as a 'GitHub showcase project'. Of the publicly available projects hosted on GitHub, a selected

set of projects are marked as 'showcases', to demonstrate how a project can be developed in certain domain such as game development, music, etc. [17]. Our assumption is that by selecting these GitHub projects we can start with a representative set of open source projects that enjoy popularity, and provide good examples of software development. Example of popular projects included in the GitHub showcase that we use for our analysis are: Javascript libraries such as 'AngularJS'[3] and 'npm'[4], and programming languages such as 'Go'[5], 'Rust'[6], and 'Scala'[7] to name a few.

(2) In case of proprietary projects our collaborating company (IBM) provided us a list of projects that are hosted on their private GitHub. We mine open source and proprietary projects, respectively, by using the public GitHub API, and a private API maintained by our collaborating company.

(3) On gathering this data, we employed a peer-coding strategy in collaboration with a team of 4 developers at IBM. We manually classified issues as bugs or enhancements. To identify these, we used the tags of the GitHub issues. As a coding guide, we—(a) ensured that there were no duplicates in enhancements or bugs; and (b) ensured the issue threads we followed up the developers.

Note that all the projects are hosted on GitHub. They have different start dates. We show the start dates of the proprietary and open source projects in Figure 1. It is worth noting that a majority of these projects have a history of at least one year.

3.2.1 Extracting Relevant Projects. It is important to note that projects hosted on GitHub gives researchers a tremendous opportunity to extract necessary project information such as issues, bugs, and enhancements [26] [7] [40]. Unfortunately, it is possible that many of these projects can contain very short development activity, can be used for personal use, or not be related to software development at all [26] [7]. These projects may bias our findings. Hence, we implement a set of rules to identify and discard these projects. We call these set of rules "filters" and they are designed such that only the projects that contain sufficient software development data for analysis pass this filter.

As the first step of filtering, we identify projects that contain sufficient software development information using the following criteria. These criteria address the limitations of mining GitHub projects as highlighted by prior researchers [26] [7]. We summarized these into the rules listed below:

- *Collaboration*: Number of pulls requests are indicative of collaboration, and the project must have at least one pull request.
- *Commits*: The project must contain more than 20 commits.
- *Duration*:The project must contain software development activity of at least 50 weeks.
- *Issues*: The project must contain more than 10 issues.
- *Personal Purpose*:The project must not be used and maintained by one person. The project must have at least eight contributors.
- *Releases*:The project must have at least one release.

[3]https://GitHub.com/angular/angular.js
[4]https://GitHub.com/npm/npm
[5]https://GitHub.com/golang/go
[6]https://GitHub.com/rust-lang/rust
[7]https://GitHub.com/scala/scala

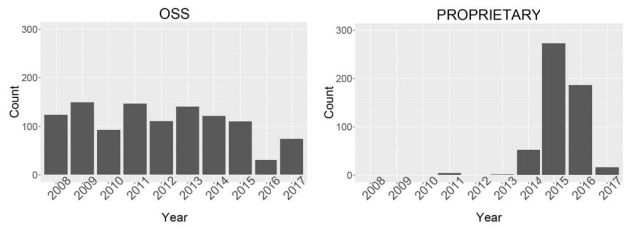


Figure 1: The project count and their start years. On the left, we plot the counts of open source projects started in each year. On the right, we count of proprietary projects started in each year.

Sanity check	Discarded project count	
	Proprietary	open-source
Collaboration (Pull requests > 0)	35	54
Commits > 20	68	96
Duration > 1 year	12	46
Issues > 10	60	89
Personal purpose (# programmers > 8)	47	67
Releases >0	136	44
SW development only	9	51
Projects Discarded	367	447
Total Projects	538	1108
Projects after filtering	171	661

Figure 2: Count of projects that pass the filter. Upon completion, we are left with 171 proprietary and 661 open-source projects.

- *Software Development*:The project must only be a placeholder for software development source code.

After applying the aforementioned filter, from our initial pool of 1,108 open source projects and 538 proprietary we were left with 661 open source and 171 proprietary projects. For details of how many projects passed each of our filter rules see Figure 2.

From Figure 2 we observe that 59.6% of the GitHub showcase projects pass the recommended sanity checks by researchers. The 447 projects filtered by applying the filter further emphasizes the need to validate software project data mined from GitHub before use lest they skew the findings.

3.3 Time Series Modeling

Autoregressive Integrated Moving Average (ARIMA) models were proposed by Box and Jenkins [8] in 1976. They are now most commonly used to model time series data to forecast the future values. The ARIMA model extends ARMA (Autoregressive Moving Average) model by allowing for non-stationary time series to be modeled, i.e., a time series whose statistical properties such as mean, variance, etc. are not constant over time.

A time series is said to be autoregressive moving average (ARMA) in nature with parameters (p, q), if it takes the following form:

$$y_t = \sum_{i=1}^{p} \phi_i y_{t-i} + \sum_{i=1}^{q} \theta_i \epsilon_{t-i} + \epsilon_t \quad (1)$$

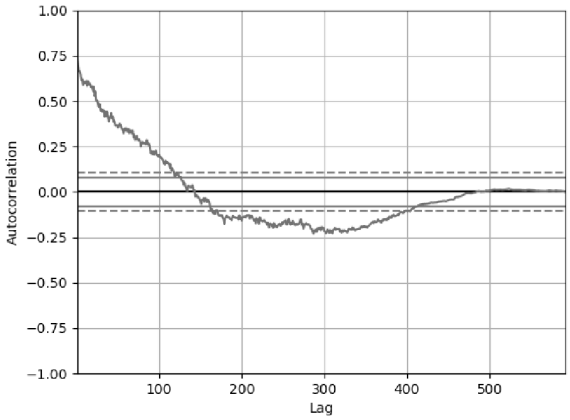


Figure 3: Autocorrelation plot of issue reports. The "lag" parameter indicates the number of weeks over which the autocorrelation was computed. Note that autocorrelation is significantly large for lag≤20 weeks.

Where y_t is the current stationary observation, y_{t-i} for $i = 1, ..., p$ are the past stationary observations, ϵ_t is the current error, and ϵ_{t-i} for $i = 1, ..., q$ are the past errors. If this original time series $\{z_t\}$ is non-stationary, then d differences can be done to transform it into a stationary one $\{y_t\}$. These differences can be viewed as a transformation denoted by $y_t = \nabla^d z_t$, where $\nabla^d = (1-B)^d$ where B is known as a backshift operator. When this differencing operation is performed, it converts an ARMA (Autoregressive Moving Average) model into an ARIMA (Autoregressive Moving Integrated Average) model with parameters (p, q, d).

Before using ARIMA, the observed time series has to be analyzed to select the parameters for $ARIMA(p, q, d)$. This requires the identification of the p, d, and q parameters. To do this, we take the following steps

(1) *Estimating p:* The value of p can be estimated by analyzing the *autocorrelation* plot of the time series. This is a plot of correlation between the time series and the same time series separated by a given interval (called *lag*). To demonstrate this procedure, consider an example autocorrelation plot of the ArangoDB project in Figure 3. In this figure, we see that the autocorrelation is significantly large for values of $lag \leq 20$. So, we may set any $p < 20$ for a good model.
(2) *Estimating d:* The value of d has to be set taking into account whether the time series is stationary or not. If the time series is stationary the we may set $d = 0$, otherwise we set $d > 0$. To determine if a time series is stationary, we use the Dickey and Fuller test [13].
(3) *Estimating q:* The value of q can be set taking into account whether the time series measurements have error in measurements. In our case, since we mine the GitHub repositories with their official API, there are no measurement errors. Thus, we set $q = 0$.

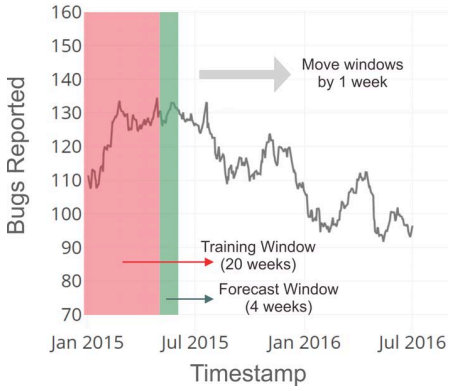


Figure 4: Rolling window approach for forecasting issues, bugs, and enhancements over time. The input window has a duration of 20 weeks (≈ 4 months) and the forecast window is 4 weeks (≈ 1 month). The step length of the moving window is 1 week.

After the ARIMA model has been constructed, we need to ensure that the given data can be accurately model by it. Like most time series modeling techniques, ARIMA has some inherent assumptions. It is therefore required that the data be preprocessed and the assumptions be satisfied before the ARIMA model is applied lest we risk inaccurate forecast. In the following, we list the preprocessing steps taken by us:

(1) *Ensuring Normality:* ARIMA model assumes that the given time series data is approximately normally distributed. In order to ensure this, the data was transformed to approximate normal distribution using power transformations.
(2) *Ensuring Stationarity:* It is assumed that the time series has a constant mean, variance, and no trend over time. We use the Dickey and Fuller test [13] to test for stationarity. If we note that the series is non-stationary, we transform using using differences. This can be achieved by setting appropriate d value in $ARIMA(p, d, q)$.

Investigating whether the given time series data satisfies these assumptions is a critical task, because falling to satisfy the assumptions leads to selecting incorrect ARIMA model. In this work, we independently verified these assumptions and applied the necessary corrective transformations before applying the ARIMA model. Additionally, we performed extensive empirical evaluations to determine the values of p, q, and d to be used in $ARIMA(p, q, d)$.

3.4 Proposed Forecasting Approach

In evaluating the performance of time series modeling, we used a rolling window time series forecasting approach as shown in Figure 4. This approach works as follows:

(1) First, we create two windows (labeled training window and testing window in Figure 4). After extensive empirical evaluation of all the projects, we determined the best training window size to be around 20 weeks (≈ 4 months) and test window as set to 4 weeks (≈ 1 month).

(2) Next, we train an ARIMA model on the time series data from the training window and forecast for issues, bugs, and enhancements over the duration of the test window.
(3) Then, we estimate the magnitude of average error (also known as MAE, described in §3.5) of the forecast values.
(4) Finally, we move the training and testing window by 1 time step (in our case this is 1 week) and repeat steps 1, 2, and 3 until we reach the end of the time series.

After the rolling widow approach described above terminates, we gather the errors in forecast (measured as MAE) and compute the mean MAE values and the spread (variance) of the MAE values. These values are plotted for all the projects as shown in Figures 5 and 7.

3.5 Measuring Forecast Error

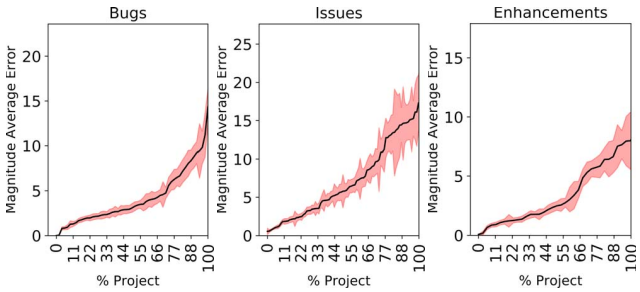
To evaluate the quality of the ARIMA models used for forecasting, we compute the mean absolute error (MAE). MAE has been a preferred method to evaluate errors in time series analysis by researchers in several areas [51, 52]. MAE is a measure of difference between two continuous variables. Assume X and Y are variables of paired observations that express the same phenomenon. Examples of Y versus X include comparisons of predicted versus observed, subsequent time versus initial time, and one technique of measurement versus an alternative technique of measurement. Consider a scatter plot of n points, where point i has coordinates (xi, yi). Mean Absolute Error (MAE) is the average vertical distance between each point and the Y=X line, which is also known as the One-to-One line. MAE is also the average horizontal distance between each point and the Y=X line. The Mean Absolute Error is given by:

$$MAE = \sum_{i=1}^{N} P_i \left| \hat{Y}_i - Y_i \right| = \sum_{i=1}^{N} P_i \left| e_i \right| \quad (2)$$

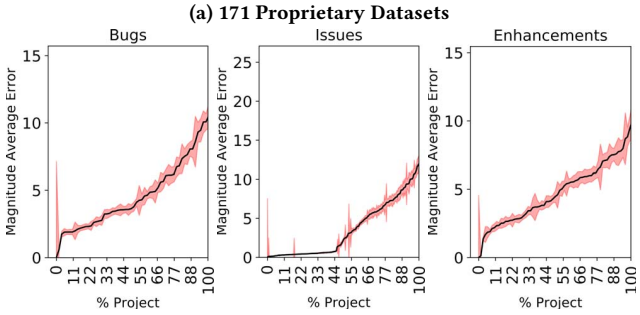
Here, N represents the total number of unique values of issues, bugs, enhancements, etc. P_i represents the probability of appearance of each of the unique values of issues, bugs, enhancements, etc. Y_i denotes the actual value, $\hat{Y}_i$ denotes the predicted values. Accordingly, lower values of MAE are considered to be better.

Some researchers [2] have endorsed the use other metrics such as mean squared error (MSE) or mean magnitude of relative error (MMRE) to measure the quality of forecast in time series models. We, however, refrain from using these measures because:

- MAE has a clear interpretation as the average absolute difference between Y_i and $\hat{Y}_i$. Many researchers [51, 52] find this measure desirable because its interpretation is clear. However, researchers frequently compute and misinterpret the Root Mean Squared Error (RMSE), which is not the average absolute error [51, 52].
- In cases where the true value is close to or equal to zero, measures like MMRE fail to provide an accurate description of the error. When true values are zero, MMRE is extremely large; this skews the errors and risks leading to spurious interpretation of the errors.



(a) 171 Proprietary Datasets



(b) 671 Opensource datasets

Figure 5: This figure verifies the existence of temporal trends in issues, bugs, and enhancements. The low magnitude of average error (MAE) values show that time series forecasting with ARIMA can be performed on these attributes.

4 EXPERIMENTAL RESULTS

RQ1: Are there temporal trends in our data?

The first research question seeks to establish the presence of temporal trends in issues, bugs, and enhancements. If temporal trends do exist in these attributes, a time series model such as ARIMA, which is equipped to make accurate forecasts, should lead to low errors when we use past data to forecast for the future. For this purpose, we ask if we may construct a time series model using past data of issues, bugs, and enhancements. That is, we attempt to: (a) forecast for future the number of bugs using past trends in bug reports, (b) forecast for future the number of issue reports using past trends in issue reports, and (c) forecast for future the number of enhancements using past trends in enhancements.

For experimentation, in each project we use a rolling window method to train an ARIMA model on past 20 weeks and forecast for future 4 weeks. This is repeated for issues, bugs, and enhancements. Then, we compute the magnitude of absolute error (MAE) between actual and forecast values for each step of the rolling window.

Our results are shown in Figure 5. For proprietary projects, in terms of MAE, the errors for forecasting bugs in 66% of the projects are very small (they are close to zero in several cases). Further, we note that the variance of these errors shaded in pink in Figure 5 are also quite low. For opensource projects, we notice similar trends. However, in case of open source projects, the MAE scores are slightly higher when compared to proprietary projects. This means that temporal trends do exist in open source projects, but

Figure 6: Spearman's ρ. Figure 6a shows the correlations in proprietary projects. Figure 6b shows the correlations in opensource projects.

these are less temporal as compared to proprietary projects. In summary, we answer this research question as follows:

> ***Lesson 1***
> The mined attributes of both proprietary and opensource projects exhibit temporal trends. Proprietary projects are slightly more temporal compared to opensource projects.

RQ2: Are there correlations between mined attributes?

Having established that the attributes we mined (issues, bugs, and enhancements) exhibit temporal trends, in this research question we explore the correlations between mined attributes. This research question was partly motivated by the findings of Ayari et al. [3], they report that, *"...at least half of all the issues in a tracking system are related to bugs and the other half contains a mix of preventive, perfective and corrective requests."* This work was published in 2007, and since then there has been a wide adoption of version control systems such as GitHub by several projects. These version control systems have integrated issue tracking mechanisms such as GitHub issues. Therefore, in this research question, we revisit this claim to check for the relationship between issues, bugs, and enhancements.

For each of our 832 proprietary and opensource projects, we compute the correlation between all pairs of attributes. For this, we used Spearman's ρ. That is, we compute correlations between:

- *issues* ↔ *bugs*
- *enhancements* ↔ *bugs*
- *issues* ↔ *enhancements*

Figure 6 shows a heatmap with the correlation values. In this figure, a value close to 1 would indicate a strong positive correlation, a value close to −1 would indicate a strong negative correlation, and a value close to 0 would indicate no correlation. A ρ value between 0.3 and 0.7 is considered moderate to strong [45]. Our findings corroborate the report of Ayari et al. We note the following:

(1) In proprietary projects, there exist two moderate to strong correlations: (a) Our strongest correlation exists between issues ↔ bugs, and (b) This is followed by issues ↔ enhancements.
(2) In opensource projects, there still exists the same two correlations. But, these correlations are relatively weaker compared to same correlations in proprietary projects.

We summarize our findings as follows:

> ***Lesson 2***
> There exist moderately strong correlations between *issues* ↔ {*bugs*, *enhancements*}. These correlations are stronger in proprietary projects compared to opensource projects.

RQ3: Can issues forecast for future bugs and enhancements?

From RQ2, we learn that there exists correlations between issues ↔ {bugs, enhancements}. In this research question, we seek to leverage those correlations to establish if it is possible to construct an ARIMA model on issues (labeled *ISSUES*) and use that model to forecast for bugs and enhancements.

For experimentation, we used a rolling window to create an ARIMA model on issues, then with that model we forecast for bugs and enhancements (this approach is described in detail in §3.4). The use of rolling window results in forecasts for each time step progressed by the window. For these time steps, we measure the error between the forecast values bugs (and enhancements) and the actual values of bugs (and enhancements) using magnitude of average error (MAE, see Equation 2). Figure 7 shows the magnitude of average error for bugs and enhancements in opensource and proprietary projects.

In order to see if the ARIMA model constructed using issues captures trends in bugs and enhancements, we plot the actual and forecast values of bugs (and enhancements) a function of bug (and enhancement) counts. We would expect an increase in the forecast of bug counts (or enhancement counts) as the actual values increase. These are shown in Figure 7 as well.

Our findings are summarized as follows:

(1) For most projects (both proprietary and opensource) the magnitude of average error is very small.
(2) The magnitude of average errors in proprietary projects are slightly lower than in opensource projects.
(3) The variance of these errors (which results from using the rolling window) are shaded in pink are also noticeably low.

These results are very encouraging indeed and we answer this research question as follows:

> ***Lesson 3***
> We find that ARIMA models built on issues can be accurate for forecasting bugs and enhancements for both proprietary and opensource projects. The errors are very low (close to zero) and the variance in errors are also significantly low.

RQ4: Are the forecasts using issues better than with past temporal data?

In RQ1, we established that ARIMA model built on past bug and enhancement data (called *LABELED*) can forecast for future bugs and enhancements with very low errors (see Figure 5). Further, in RQ3, we showed that an ARIMA model can built on past issue data (called *ISSUE*) can also forecast for future bugs and enhancements with low magnitudes of average errors (see Figure 7). This research question is a natural extension of those two results. Here we compare errors in forecasts obtained from *ISSUE* and *LABELED* where:

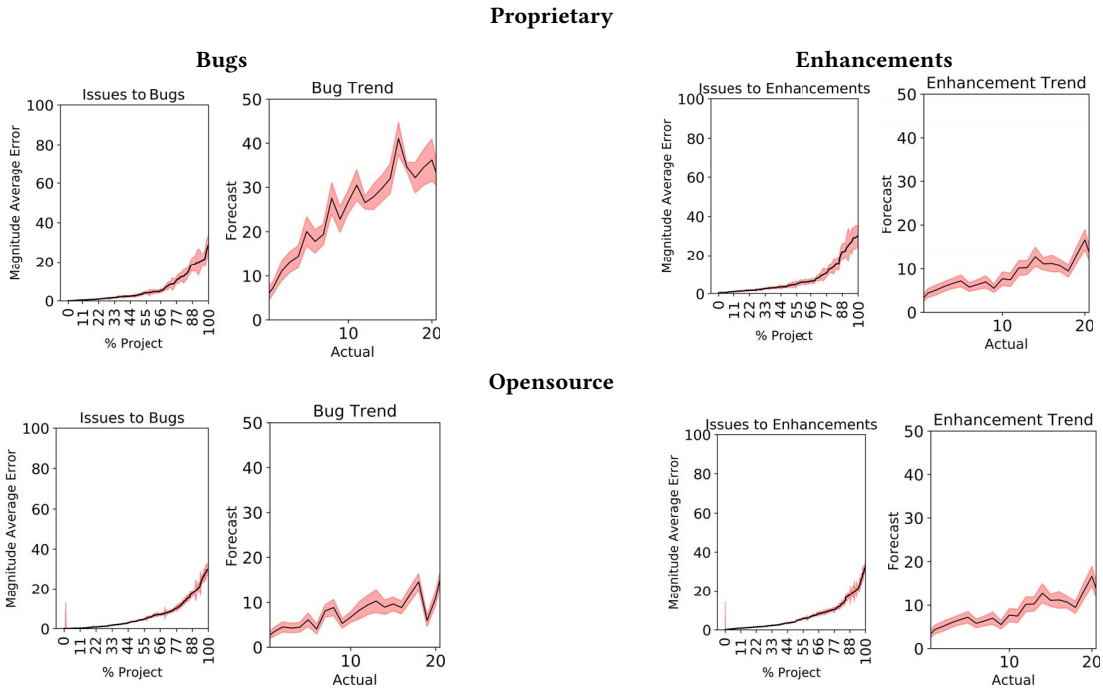
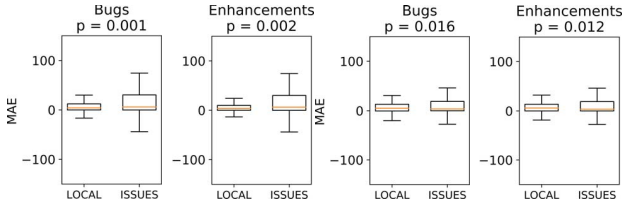


Figure 7: This figure shows forecasts for future bugs and enhancements using ARIMA models constructed using past issue reports. For both proprietary and opensource projects, the magnitude of average error is very low (close to zero in several cases). Trend graphs show that an increase in actual bugs (or enhancements) leads to a corresponding increase in forecasts.



(a) Proprietary **(b) Opensource**

Figure 8: Compare distribution of errors with ARIMA forecasts using issue reports (*ISSUE*) and past temporal data (*LABELED*). The charts indicate that the distribution of errors are very similar to each other with p-values < 0.05.

a) *ISSUE* : ARIMA model built using past issue report trend.
b) *LABELED* : ARIMA model built using past bug report trends and past enhancement request trends.

To perform this comparison, we use both of these models to forecast for future bugs and enhancements. The construction of *LABELED* model is described RQ1 and RQ3 describes the construction of *ISSUE*. We compute the error in forecasts with *ISSUE* and *LABELED* and use a parametric statistical hypothesis test (Welch's t-test) to compare the errors[8]. In order to conduct the hypothesis test we use the following hypothesis:

$\mathcal{H}$: The distributions of errors in using *ISSUE* is significantly larger than errors in using *LABELED*.

If the *p value* of the above hypothesis is less than 0.05, then we may reject that hypothesis and assert that, *"the distributions of errors in using ISSUE is statistically similar to errors in using LABELED."*

The distribution of errors for forecasting bugs and enhancements in proprietary and opensource projects are shown Figure 8. It is immediately noticeable the expected value of the errors are close to zero in both *ISSUE* and *LABELED* for all cases. Additionally, the *p values* are always less than 0.05 in all the cases. Therefore, the answer to this research question is:

Lesson 4

In the 832 projects studied here, forecasts made using past temporal data statistically comparable to forecasts made using only the issue reports.

Note that this is a result of much practical importance since the effort involved in building the issue models of **RQ3** is much lower than the bug (and enhancement) models seen in **RQ1**. The comparable errors indicate that it is not necessary to identify bugs or enhancements separately from issues to forecast for their future values. Rather, we may simply mine for issues and use that to forecast for future bugs and enhancements. Doing this significantly reduces effort required to mine for each one individually.

[8]We use a parametric test because it is known that the errors of ARIMA models have a normal distribution [8]

5 THREATS TO VALIDITY

This work, like any other empirical study, is subject to the following threats to validity.

Model Bias: For building the time series models, in this study, we elected to use ARIMA. We chose this method because past studies show that, for time series modeling, the results were superior to other methods. Other time series modeling methods like long short term memory (LSTM) models have shown promise in other areas. That comparison is beyond the scope of this paper.

Evaluation Bias: This paper uses one measure of error, MAE (see Equation 2). Other quality measures often used in software engineering to quantify the effectiveness of forecast. A comprehensive analysis using these measures is left for future work.

Sample bias: Our data was gathered by mining GitHub. Several data sets may often be noisy and uninteresting. Further, all the issues were manually peer-coded as bugs or enhancements. Although the coding policy was consistent among peers, these may still lead to some biases. To address these, we take the following steps: (1) Apply sanity checks to filter out irrelevant projects (§3.2); (2) Include a wide variety of projects; (3) Report mean and standard errors in all our measurements; and (4) Perform statistically sound comparisons when appropriate (e.g. in RQ4 we use Welch's t-test).

Further, we have discussed our findings with business users who are software engineers and managers at IBM. They agreed with the general direction of our findings: they stated that issues can be used as indicators of future bugs and enhancements. They also agreed that there are differences between open source and proprietary software development, and we should not assume these tools and techniques will help the practitioners of interest, in the same manner.

6 DISCUSSION

A key motivation of this work is to reduce the amount of unexpected work assigned to any developer in any month. Developers are frequently reassigned to different projects. It is well established that a person who works on more than one project at a time incurs a cost in terms of time required to change contexts at each shift from one project to the other. The more complex the task, the more time it takes to make the shift [47]. Gerald Weinberg [50] showed that for software projects, the cost of switching between projects escalated if each task has a even a 10% penalty as shown in Figure 9. In real world, the costs are usually much higher.

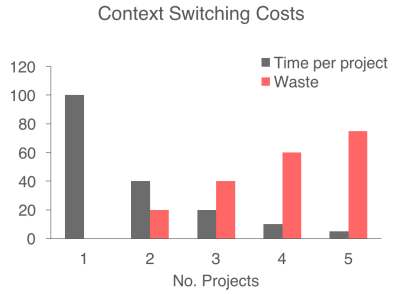


Figure 9: Cost of switching contexts. Image courtesy of [50].

For large organizations that maintain their software suites based on end-user issue reports, it is vitally important for managers not to overload staff with tasks. It may be beneficial to plan weeks ahead in order to even out the workflow amongst all developers and alleviate effort involved in switching contexts. Of course, for that to work, we need a planning agent that can forecast the future. The results of this paper show that —

> *Managers need to only track issue trends in projects. With these trends they may be able to forecast the number of future bugs reports and enhancement requests.*

We note that, this result would very useful for large service organizations maintaining suites of software attempting to effectively manage personnel across projects.

7 CONCLUSION AND FUTURE WORK

In summary, we mined 832 projects (661 opensource and 171 proprietary projects) for issue reports over time. Following this, we spent over two months curating these issues into bugs and enhancements. The effort involved in the curation process lead us to investigate if these attributes are correlated with each other. We discovered that the attributes were indeed correlated and that it is possible to build time series models on one attribute (issues) and use that to forecast for bugs and enhancements. Our method can be used to circumvent much of the complex machinery required to commission and maintain a convention bug predictors. In addition to the simplicity, we show that our method is quite accurate (with near zero errors) in more than 66% of the projects explored here. For future work, we believe a detailed exploration is warranted to identify why the given approach is ineffective in the remaining 34% of the projects.

Our claim is not that issues provide the best forecast for bugs (or enhancements), rather it is that issue trends may be leveraged to supplement bugs forecast with a sufficiently high degree of accuracy. And these forecasts may be very useful for anticipating the required managerial actions. However, it must be considered that even though a project manager could e.g., allocate two more developers to a project when our approach forecasts several bug reports, an interesting future work direction would be to assist in identifying which developers more precisely, perhaps based on their experiences. For example, the manager may be able to allocate a developer that works on databases to a project because of the forecast of bugs emerging in another project related to databases.

While our results are biased by our sample of projects, we have made attempts to include a wide array of projects both proprietary and opensource. Our work highlights the merits of mining software repositories to perform time series analysis. To the best of our knowledge, there exists no large study that reports the opposite of our conclusions.

REFERENCES

[1] Pekka Abrahamsson, Outi Salo, Jussi Ronkainen, and Juhani Warsta. 2017. Agile software development methods: Review and analysis. *arXiv preprint arXiv:1709.08439* (2017).

[2] Ayman Amin, Lars Grunske, and Alan Colman. 2013. An approach to software reliability prediction based on time series modeling. *Journal of Systems and Software* 86, 7 (2013), 1923–1932.

[3] Kamel Ayari, Peyman Meshkinfam, Giuliano Antoniol, and Massimiliano Di Penta. 2007. Threats on building models from cvs and bugzilla repositories: the mozilla case study. In *Proceedings of the 2007 conference of the center for advanced studies on Collaborative research*. IBM Corp., 215–228.

[4] CG Bai, QP Hu, Min Xie, and Szu Hui Ng. 2005. Software failure prediction based on a Markov Bayesian network model. *Journal of Systems and Software* 74, 3 (2005), 275–282.

[5] May Barghout, Bev Littlewood, and Abdalla Abdel-Ghaly. 1998. A non-parametric order statistics software reliability model. *Software Testing Verification and Reliability* 8, 3 (1998), 113–132.

[6] Andrew Begel and Nachiappan Nagappan. 2007. Usage and perceptions of agile software development in an industrial context: An exploratory study. In *Empirical Software Engineering and Measurement, 2007. ESEM 2007. First International Symposium on*. IEEE, 255–264.

[7] Christian Bird, Alex Gourley, Prem Devanbu, Michael Gertz, and Anand Swaminathan. 2006. Mining email social networks. In *Proceedings of the 2006 international workshop on Mining software repositories*. ACM, 137–143.

[8] George EP Box, Gwilym M Jenkins, Gregory C Reinsel, and Greta M Ljung. 2015. *Time series analysis: forecasting and control*. John Wiley & Sons.

[9] Bernard Burtschy, Grigore Albeanu, Dragos N Boros, Florin Popentiu, and Victor Nicola. 1997. Improving software reliability forecasting. *Microelectronics Reliability* 37, 6 (1997), 901–907.

[10] S Chatterjee, RB Misra, and SS Alam. 1997. Prediction of software reliability using an auto regressive process. *International journal of systems science* 28, 2 (1997), 211–216.

[11] CKS Chong Hok Yuen. 1988. On analyzing maintenance process data at the global and the detailed levels: A case study. In *Proceedings of the IEEE Conference on Software Maintenance*. 248–255.

[12] Márcio das Chagas Moura, Enrico Zio, Isis Didier Lins, and Enrique Droguett. 2011. Failure and reliability prediction by support vector machines regression of time series data. *Reliability Engineering & System Safety* 96, 11 (2011), 1527–1534.

[13] David A Dickey and Wayne A Fuller. 1979. Distribution of the estimators for autoregressive time series with a unit root. *Journal of the American statistical association* 74, 366a (1979), 427–431.

[14] Tore Dybå and Torgeir Dingsøyr. 2008. Empirical studies of agile software development: A systematic review. *Information and software technology* 50, 9 (2008), 833–859.

[15] N Fenton, Martin Neil, and D Marquez. 2008. Using Bayesian networks to predict software defects and reliability. *Proceedings of the Institution of Mechanical Engineers, Part O: Journal of Risk and Reliability* 222, 4 (2008), 701–712.

[16] Eduardo Fuentetaja and Donald J Bagert. 2002. Software evolution from a time-series perspective. In *Software Maintenance, 2002. Proceedings. International Conference on*. IEEE, 226–229.

[17] Github. [n. d.]. Build software better, together. ([n. d.]). https://github.com/showcases

[18] Michael Godfrey and Qiang Tu. 2001. Growth, evolution, and structural change in open source software. In *Proceedings of the 4th international workshop on principles of software evolution*. ACM, 103–106.

[19] Amrit L. Goel. 1985. Software reliability models: Assumptions, limitations, and applicability. *IEEE Transactions on software engineering* 12 (1985), 1411–1423.

[20] Amrit L Goel and Kazu Okumoto. 1979. Time-dependent error-detection rate model for software reliability and other performance measures. *IEEE transactions on Reliability* 28, 3 (1979), 206–211.

[21] Israel Herraiz, Jesus M Gonzalez-Barahona, and Gregorio Robles. 2007. Forecasting the number of changes in Eclipse using time series analysis. In *Mining Software Repositories, 2007. ICSE Workshops MSR'07. Fourth International Workshop on*. IEEE, 32–32.

[22] SL Ho and M Xie. 1998. The use of ARIMA models for reliability forecasting and analysis. *Computers & industrial engineering* 35, 1-2 (1998), 213–216.

[23] Harold Edwin Hurst. 1951. Long-term storage capacity of reservoirs. *Trans. Amer. Soc. Civil Eng.* 116 (1951), 770–808.

[24] Z Jelinski and PB Moranda. 1972. Software reliability research, Statistical Computer Performance Evaluation, W. Freiberger (ed.), 465–484. (1972).

[25] Guo Junhong, Liu Hongwei, and Yang Xiaozong. 2005. An autoregressive time series software reliability growth model with independent increment. In *Proceedings of the 7th WSEAS International Conference on Mathematical Methods and Computational Techniques In Electrical Engineering*. World Scientific and Engineering Academy and Society (WSEAS), 362–366.

[26] Eirini Kalliamvakou, Georgios Gousios, Kelly Blincoe, Leif Singer, Daniel M German, and Daniela Damian. 2014. The promises and perils of mining github. In *Proceedings of the 11th working conference on mining software repositories*. ACM, 92–101.

[27] Nachimuthu Karunanithi, Darrell Whitley, and Yashwant K. Malaiya. 1992. Using neural networks in reliability prediction. *IEEE Software* 9, 4 (1992), 53–59.

[28] Chris F. Kemerer and Sandra Slaughter. 1999. An empirical approach to studying software evolution. *IEEE Transactions on Software Engineering* 25, 4 (1999), 493–509.

[29] Benedicte Kenmei, Giuliano Antoniol, and Massimiliano Di Penta. 2008. Trend analysis and issue prediction in large-scale open source systems. In *Software Maintenance and Reengineering, 2008. CSMR 2008. 12th European Conference on*. IEEE, 73–82.

[30] N Raj Kiran and Vadlamani Ravi. 2008. Software reliability prediction by soft computing techniques. *Journal of Systems and Software* 81, 4 (2008), 576–583.

[31] Rahul Krishna, Tim Menzies, and Wei Fu. 2016. Too much automation? The bellwether effect and its implications for transfer learning. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. ACM, 122–131.

[32] Rahul Krishna, Tim Menzies, and Lucas Layman. 2017. Less is more: Minimizing code reorganization using XTREE. *Information and Software Technology* 88 (2017), 53–66.

[33] Manny M Lehman and Laszlo A Belady. 1985. *Program evolution: processes of software change*. Academic Press Professional, Inc.

[34] Jung-Hua Lo. 2009. The implementation of artificial neural networks applying to software reliability modeling. In *Control and Decision Conference, 2009. CCDC'09. Chinese*. IEEE, 4349–4354.

[35] Michael R Lyu. 2007. Software reliability engineering: A roadmap. In *2007 Future of Software Engineering*. IEEE Computer Society, 153–170.

[36] Michael R Lyu et al. 1996. Handbook of software reliability engineering. (1996).

[37] Tim Menzies, Jeremy Greenwald, and Art Frank. 2007. Data mining static code attributes to learn defect predictors. *IEEE transactions on software engineering* 33, 1 (2007), 2–13.

[38] Tim Menzies, Zach Milton, Burak Turhan, Bojan Cukic, Yue Jiang, and Ayşe Bener. 2010. Defect prediction from static code features: current results, limitations, new approaches. *Automated Software Engineering* 17, 4 (2010), 375–407.

[39] Subhas Chandra Misra, Vinod Kumar, and Uma Kumar. 2009. Identifying some important success factors in adopting agile software development practices. *Journal of Systems and Software* 82, 11 (2009), 1869–1890.

[40] Nuthan Munaiah, Steven Kroh, Craig Cabrey, and Meiyappan Nagappan. 2017. Curating GitHub for engineered software projects. *Empirical Software Engineering* 22, 6 (2017), 3219–3253.

[41] Martin Neil and Norman Fenton. 1996. Predicting software quality using Bayesian belief networks. In *Proceedings of the 21st Annual Software Engineering Workshop*. NASA/Goddard Space Flight Centre, 217–230.

[42] Ping-Feng Pai and Wei-Chiang Hong. 2006. Software reliability forecasting by support vector machines with simulated annealing algorithms. *Journal of Systems and Software* 79, 6 (2006), 747–755.

[43] Ping-Feng Pai and Wei-Chiang Hong. 2006. Software reliability forecasting by support vector machines with simulated annealing algorithms. *Journal of Systems and Software* 79, 6 (2006), 747–755.

[44] David G Robinson and Duane Dietrich. 1987. A new nonparametric growth model. *IEEE Transactions on Reliability* 36, 4 (1987), 411–418.

[45] George AF Seber and Alan J Lee. 2012. *Linear regression analysis*. Vol. 936. John Wiley & Sons.

[46] Nozer D. Singpurwalla and Refik Soyer. 1985. Assessing (software) reliability growth using a random coefficient autoregressive process and its ramifications. *IEEE Transactions on Software Engineering* 12 (1985), 1456–1464.

[47] Alina Tugend. 2008. Multitasking Can Make You Lose ... Um ... Focus. (Oct 2008). https://nyti.ms/2jD6gzj

[48] Burak Turhan, Tim Menzies, Ayşe B Bener, and Justin Di Stefano. 2009. On the relative value of cross-company and within-company data for defect prediction. *Empirical Software Engineering* 14, 5 (2009), 540–578.

[49] Wladyslaw M Turski. 1996. Reference model for smooth growth of software systems. *IEEE Transactions on Software Engineering* 22, 8 (1996), 599.

[50] Gerald M Weinberg. 1992. Quality software management (Vol. 1): systems thinking. (1992).

[51] Cort J Willmott and Kenji Matsuura. 2005. Advantages of the mean absolute error (MAE) over the root mean square error (RMSE) in assessing average model performance. *Climate research* 30, 1 (2005), 79–82.

[52] Cort J Willmott and Kenji Matsuura. 2006. On the use of dimensioned measures of error to evaluate the performance of spatial interpolators. *International Journal of Geographical Information Science* 20, 1 (2006), 89–102.

[53] MP Wiper, AP Palacios, and JM Marín. 2012. Bayesian software reliability prediction using software metrics information. *Quality Technology & Quantitative Management* 9, 1 (2012), 35–44.

[54] Alan Wood. 1997. Software reliability growth models: assumptions vs. reality. In *Software Reliability Engineering, 1997. Proceedings., The Eighth International Symposium on*. IEEE, 136–141.

[55] Jingwei Wu and Richard Holt. 2006. Seeking empirical evidence for self-organized criticality in open source software evolution. (2006).

[56] M Xie and SL Ho. 1999. Analysis of repairable system failure data using time series models. *Journal of Quality in Maintenance Engineering* 5, 1 (1999), 50–61.

[57] Bo Yang, Xiang Li, Min Xie, and Feng Tan. 2010. A generic data-driven software reliability model with model mining technique. *Reliability Engineering & System Safety* 95, 6 (2010), 671–678.

[58] S Jamal H Zaidi, Syed Nasir Danial, and Bilal A Usmani. 2008. Modeling inter-failure time series using neural networks. In *Multitopic Conference, 2008. INMIC 2008. IEEE International*. IEEE, 409–411.

[59] David Zeitler. 1991. Realistic assumptions for software reliability models. In *Software Reliability Engineering, 1991. Proceedings., 1991 International Symposium on*. IEEE, 67–74.