Using Time Series Models for Defect

Prediction in Software Release Planning

James Tunnell and John Anvik

Computer Science Department

Central Washington University

Ellensburg, WA 98926, USA

[tunnellj, janvik]@cwu.edu

*Abstract*—To produce a high-quality software release, sufficient time should be allowed for testing and fixing defects. Otherwise, there is a risk of a slip in the schedule and/or the quality. To this end, this paper presents a time series model using historical information for predicting the number of defects for the next product release based on hypothetical values for features and improvements completed in the next release. This allows for hypothetical release plans to be compared to assess their predicted impact on testing and defect-fixing time.

Keywords-software; defect; quality; release planning; testing; prediction; time-series;

#### Introduction

There are two primary concerns in software release planning: improving functionality and maintaining high quality. Both objectives are constrained by limits on development time and budget. To respect these constraints and meet both objectives, the scope of the planned work must be limited, such that there is time available to properly handle the inevitable defects (bugs) that will arise. In this way, a high quality of software product can be produced while also improving its functionality.

A significant consideration in the release planning process is the amount of time allocated for testing and bug-fixing. If this factor is not considered, the project risks a slip in the schedule or the quality of the product. As the time required for testing and bug-fixing will likely be a function of the number of defects introduced during development, it is desirable to be able to predict how many bugs can be expected as development proceeds.

Defect prediction techniques generally fall into two categories; those based on code analysis and those based on statistical analysis. This paper presents an alternative technique: a time series model.

A potential application of a defect prediction model is for comparing different release plans to estimate the bug fallout for a particular release plan. This would assist release planners compare release plans in ensuring that the total development time does not exceed the project’s time budget for a release. The comparison of different release plans is integral to release plan optimization. Release plan optimization is a focus of The Next Release Problem [2], a key problem in Search-Based Software Engineering (SBSE) [10, 15, 17].

.

For the defect prediction model to be useful in comparing release plans, the model must depend on the basic elements of the release plan: the features and improvements planned for the next release, and the defects from past releases. More specifically, the model uses of a multivariate time series model that includes exogenous inputs.

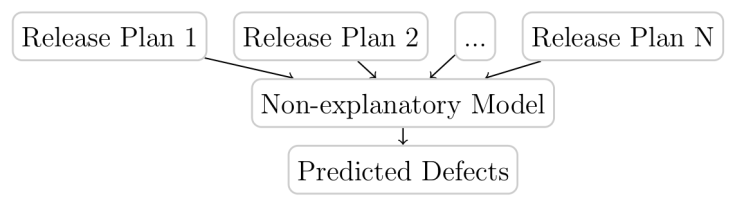
This paper proceeds as follows. First, further motivation for the use of a time-series model is presented in Section II. Next, we present some background about time series modelling in Section IV. Sections V and VI present our data and modelling methodology, respectively. We then present the result of apply the time-series modelling approach to data from the *MongoDB*[[1]](#footnote-1) software project in Section VII. The paper then concludes in Section VIII.

#### Motivation

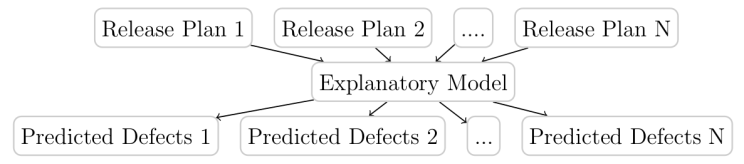
If software releases for a project are planned in a manner consistent with previous releases, it is reasonable to construct a statistical predictive model of defects that is dependent only on the occurrence of previous defects within the project. After all, planned features and improvements are likely to be selected in a manner similar to that used for previous releases, so it should be safe to assume that defect occurrences in the next release will likewise occur in same like manner as in previous releases.

This assumption is logical under normal planning conditions where planners rely on their experience and established project conventions to arrive at a satisfactory set of requirements for an upcoming release. However, if release planners instead opt to employ a heuristic or optimization to make their decision, this would require the comparison of multiple hypothetical release plans. In this case, one would expect that the predicted number of defects would vary across the release plans as the set of features and improvements would differ in each hypothetical scenario. In other words, in order for release planners to be able to create ‘what-if’ release plan scenarios, the defect prediction needs to also consider the proposed features and improvements, not just the previous defects. Such a model would assume some explanatory relationship. Fig. 1 and 2 illustrates this point.

The use of such a modelmaymore accurate means forhypothetical By improving the accuracy of defect prediction, the release planner can ein the schedule thereby maintaining a high software quality and the value or of the next release.



1. Using a non-explanatory model would result in the same defect prediction, regardless of the release plan.



1. Using an explanatory model allows for the possibility of different defect pre-dictions for each release plan.

## The Next Release Problem

Release plan optimization is the goal of The Next Release Problem (NRP), a problem shown to be NP-Hard [2]. As the NRP is abstract in its treatment of feature cost, a broad range of optimization techniques can be applied, such as integer programming, hill climbing, simulated annealing, and genetic algorithms.

The NRP describes the situation where software project planners, who have multiple customers to satisfy, would like to maximize the value, such as revenue, produced by completing a release. Briefly, the problem is posed as follows:

* A software project has some set of requirements to consider for implementation in the next release.
* A customer will provide revenue when a particular subset is included in the release.
* Dependencies may exist between requirements.
* Each requirement has an associated cost.
* The total cost is the sum of costs for all requirements included for the next release, including dependencies.
* Total cost must be below some budgeted amount.

Given this problem, the goal of the NRP is to choose the requirements subset that maximizes the total customer revenue while staying under budget.

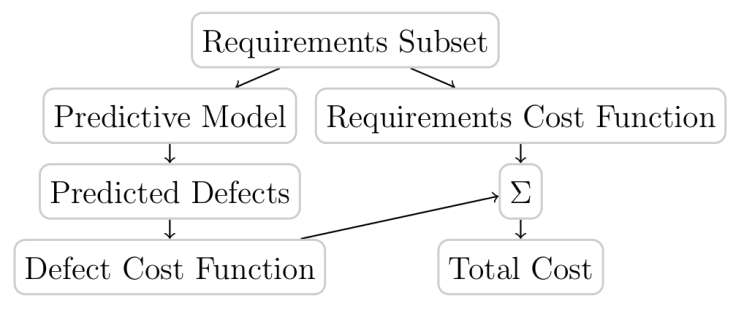
However, there exists a gap between the abstract domain of the NRP and the concrete reality of noisy data found in software projects. The application of an explanatory predictive model provides a means of bridging this gap, opening up the potential for using NRP optimization techniques in real-world release planning.

As was previously discussed, a planner would need the following information to implement a NRP-like optimization:

1. A set of candidate requirements for implementation.
2. A set of customers whose needs are satisfied by a subset of the requirements, and who have an associated weight.
3. A cost function that quantifies the cost of each requirement.
4. A cost budget that should not be exceeded.

With this set of information, a planner could proceed to optimize the subset of requirements planned for the next release. However, an obvious difficulty is the definition of a cost function. A possibility is to use the estimated time to implement a requirement as the cost. However, in order to maintain quality software, the total cost of any requirement should take into consideration which includes both the cost of implementation and the cost of fixing defects associated with the feature. If the associated defect cost is not considered, a release plan may appear to be within budget, when in reality the budget will be exceeded.

An explanatory model is used to address the consideration of defect cost. Given a subset of proposed requirements, such a model could be used to predict defects and determine the additional cost, as shown in Fig. 3.



1. Defect prediction model used in determining overall cost of some requirements subset.

Since predictive models rarely have perfect accuracy, confidence levels are an important part of any prediction. Taking into account the confidence of a prediction allows release planners to assess the risk in relying on the defect prediction. Planners can choose a narrower prediction window, in exchange for a larger risk that the prediction is inaccurate. Conversely, a wider prediction window means that the potential cost range is also wider with a lower risk of inaccuracy.

#### Time Series Modeling

In this section, the concepts and structures of time series modeling is discussed.

## Time Series

A time series is a collection of observations that occur in order. The process underlying a time series is assumed to be stochastic, so the model must correspondingly be probabilistic. Critically, the sequence of observations cannot be re-arranged, as each observation is typically dependent on one or more previous observation. It is this data dependency that complicates the modeling of time series data, otherwise the observations would be independent and a probability distribution could be used to model the data.

## Autocorrelation, ACF, and PCF

The use of autocorrelation is an important part of time series modeling. Autocorrelation measures the correlation of a sequence with itself. The autocorrelation function (ACF) and partial autocorrelation function (PACF) are used to measure autocorrelation as a function of time lag. These functions can be used to identify a time series that can be modeled by a pure autoregressive function or by a pure moving average function. ACF and PACF are also used to analyze residuals, or the difference between actual and fitted values, to check for statistically significant autocorrelation.

## ARMA and ARIMA (Univariate) Models

The Box-Jenkins methodology describes the univariate ARMA and ARIMA models. The idea for these models begins with the idea of a sequence of independent shocks, generated by “random drawings from a fixed distribution” [4]. These shocks are knowns as a white noise process. The basic autoregressive, moving average (ARMA) stochastic model is then formed by a linear combination of the previous white noise values and the previous time series outputs. In the ARMA model, the ACF and PACF produce a vector as the time series is univariate.

The ARMA stochastic model requires stationarity (or approximate stationarity). Differencing is performed to deal with data that is non-stationary. Adding differenced data leads to an extension of the ARMA model, the ARIMA model.

## Endogeneity and Exogeneity

Exogenous variables are not considered to be under the “control” of the model, and are instead considered as an input. Therefore, a model should not try to account for an exogenous variable’s behavior, but instead use past values to predict the behavior of other, endogenous (non-exogenous) variables.

## VAR and VARX (Multivariate) Models

By extending the ARMA model to the multivariate case (i.e. allowing for multiple time series), a Vector ARMA model is formed. A special case of this model is the pure autoregressive model, or Vector AR (VAR) model. By also considering exogenous variables, the model becomes VARX. The VARX model fits the needs of the situation described in the Motivation section.

However, these vector models do not account for non-stationary time series. Therefore, before VARX can be used, the time series data must be shown to be either be stationary, or can be differenced to become stationary. Trends and tests for stationarity will be discussed next.

## Trends

Trending time series are challenging to analyze, because the summary statistics of mean, variance, and autocovariance will vary over time, and are therefore not interpretable [5]. Two trend types are discussed here: deterministic and stochastic.

A deterministic trend will be moving upward or downward, meaning that the time series mean is non-constant. However, the time series will be constant according to a deterministic function and the time series movements will generally follow the deterministic function, with non-permanent fluctuations above or below. Such a time series is said to be stationary around a deterministic trend.

In contrast, a stochastic trend shows permanent effects whenever random shocks occur, and will not necessarily fluctuate only close to the area of a deterministic function. The application of differencing can be used to remove a stochastic trend. Next, tests are discussed for determining if a deterministic or stochastic trend is present.

## Stationarity Tests

Stationarity can be strict or weak (of some order). Strict stationarity occurs when the statistical properties are invariant with respect to shifts of the time origin [12]. Alternatively, a weak stationarity (of second order) can be established, and strict stationarity can be established by then assuming normality [4].

For a multivariate time series, stationarity holds if all the component univariate time series are stationary [16]. Therefore, the goal of stationarity testing is to establish second-order stationarity for each univariate time series component, and then show that the assumption of normality is reasonable, thereby establishing the stationarity of the multivariate time series as a whole.

## Unit Root and Stationarity Testing

A time series that contains a stochastic trend is non-stationary. A pure auto-regressive (AR) model of such a time series contains a unit root [5]. Testing for the presence of a unit root can therefore be used to test for non-stationarity. A unit-root test poses as the null hypothesis that an AR model has a unit root. Then, a test statistic is measured. If test statistic is found to be significant, the null hypothesis cannot be rejected, and it is established that the time series has a stochastic trend and is therefore non-stationary. The augmented Dickey Fuller (ADF) test is often used for unit root testing.

On the other hand, a stationarity test uses the null hypothesis that a time series is stationary around a deterministic trend. If the test statistic shows that this hypothesis can be rejected, at some significance level, then a stochastic trend should be considered by the unit root test. The Kwiatkowski–Phillips–Schmidt–Shin (KPSS) test can be applied for testing stationarity.

#### 

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

Time Series

#### Modeling Methodology

The typical methodology used for building time series models involves specification, estimation, and diagnostics checking [4, p. 478]. Once specified and estimated, the diagnostic checking step ensures that only valid models are considered for selection. The final step of modeling is selection, where the models are compared by some model selection criterion [4, p. 581]. This section presents our approach to specifying, estimating, diagnostics checking and model selection for defect prediction.

## Model Specification & Estimation

The specification of a model is accomplished by choosing an order , which is the number of autoregressive terms to include in the model. Once an order is specified, the model parameters can be estimated by a procedure such as least squares regression.

The model order will directly affect the number of parameters included in the model. One goal of specification will be to avoid having too many parameters relative to the number of observations. The following derivation will lead to a simple rule for limiting the model order in this respect. First, let be the number of time samples in a time series. When there are time series, each sample contains observations, so there are total observations for all time series. Next, for a model of the m time series variables, there are unknown parameters to be estimated. Let the ratio of observations to parameters be denoted by

To keep *K* at or above some minimum ratio , we form the inequality

In terms of *p* this becomes

For a fixed value of , an upper bound on the model order would be

With this upper bound, model specification will include the generation of models having order. These models, with their estimated parameters, will be candidates for final model selection after undergoing diagnostic checking.

To estimate the parameters of a VARX model, we used the *dse[[2]](#footnote-6)* library provided by the estVARXar function.

## Diagnostics Checking

Diagnostic checking is performed to verify that a model can be accepted. This step includes testing for stability and for model inadequacy.

For an ARMA model to be stable, the roots of the process characteristic equation must lie outside the unit circle [4, p. 56]. Equivalently, the inverse of the roots must lie inside the unit circle. We used stability function of the *dse* libraryto perform this test.

For an ARMA model to be accurate, it is sufficient to show that “As the series length increases, the [model residuals] become close to the white noise...” [4, p. 338]. For this reason, the model inadequacy tests are formed around a study of the residuals. These lack-of-fit tests are a kind of portmanteau test.

One of these tests, the Ljung-Box test, forms a statistic from the autocorrelation of the residuals, up to some lag. In this test, the null hypothesis is that residuals are independent, so their autocorrelation is not high enough to be distinguished from a white noise series. To support this hypothesis, the test’s p-value should be above some level of significance, say 5%. We used the Box.test function from the *stats[[3]](#footnote-7)* library for performing the Ljung-Box test.

## Model Selection

Model selection criteria are used to compare models by their fit, to minimize residual error, and to penalize the model to some degree based on the number of parameters. There are a number of different selection criteria used for ARMA models, including Akaike Information Criterion (AIC), AICc (AIC with correction), and BIC (Bayesian Information Criterion). Bisgaard and Kulahci noted that “[t]he penalty for introducing unnecessary parameters is more severe for BIC and AICC than for AIC” [3]. Awas chosen AIC is defined as:

,

where is the number of samples, is the number of parameters, and is the residual covariance matrix estimate. We used the bestTSestModel function from the *dse* library for performing model selection.

#### 

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

#### Results

## Data Collection

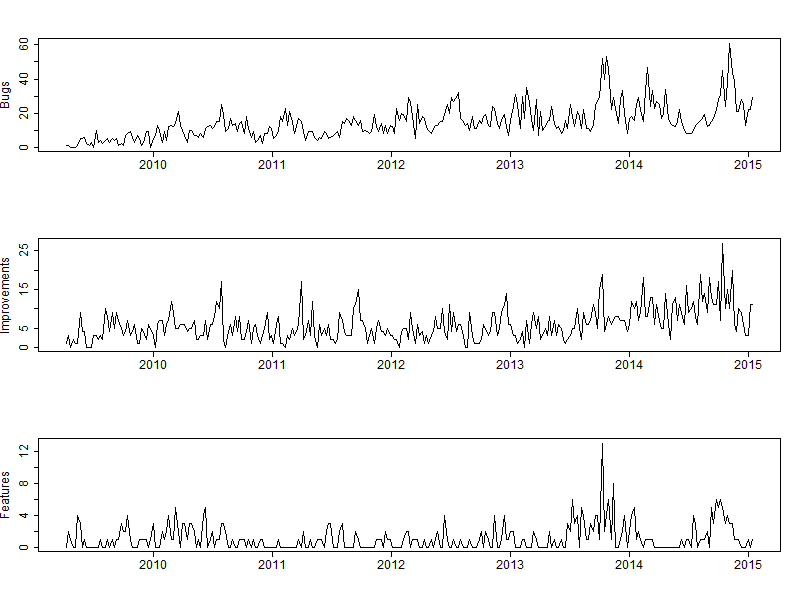
The *MongoDB* dataset was collected according to the methodology in the Data Methodology section, and the data set was sampled with a 7-day sample period to create the following time series: bugs created, improvements resolved, and new features resolved. These time series will be denoted , , and , respectively, and are shown in Fig. 5.

## Stationarity Testing

Before modeling, the time series were all checked for stationarity. The result of the ADF unit root and KPSS stationarity tests are listed in Table 2.

1. Results of running the ADF unit root test and KPSS stationarity test on , , and .

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Statistic |  | |  | |  | |
| Value | Signif. | Value | Signif. | Value | Signif. |
| ADF () | -5.020 | < 1% | -7.402 | < 1% | -7.845 | < 1% |
| ADF () | 12.65 | < 1% | 27.42 | < 1% | 30.77 | < 1% |
| KPSS | 2.852 | < 1% | 2.021 | < 1% | 0.5269 | 2.5-5% |



1. Time series data from the *MongoDB* dataset.

The unit root tests showed less than 1% significance for all time series. However, the stationarity test also showed low significance, meaning we have evidence to reject the hypothesis of stability. Since there is disagreement in the test results, the time series are differenced and the tests rerun.

After differencing we obtain the time series shown in Fig. 6, which will be referred to as , , and . Now the result of the unit root and stationarity test (listed in Table 3) both agree. That is, we can reject the hypothesis that a unit root (stochastic trend) is present at the 1% significance level and we fail to reject the hypothesis of stationarity with greater than 10% significance. Hence, the differenced time series will be used to move forward with modeling.

1. Results of running the ADF unit root test and KPSS stationarity test on , , and .

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Statistic |  | |  | |  | |
| Value | Signif. | Value | Signif. | Value | Signif. |
| ADF () | -17.65 | < 1% | -20.44 | < 1% | -21.90 | < 1% |
| ADF () | 155.8 | < 1% | 208.9 | < 1% | 239.8 | < 1% |
| KPSS | 0.0115 | > 10% | 0.0127 | > 10% | 0.0127 | > 10% |



1. Differenced time series data.

## Time Windowing

A 78-week time window (approximately 18 months) was established to restrict model scope. Three of these windowed periods, non-overlapping, were kept for modeling. Since the data is being differenced, the first sample (week) is skipped. These windowed periods are denoted *W2-79*, *W80−157*, and *W158−235*.

## Time Series Model

The model, discussed in the Time Series Modeling section, was used to model the time series. This model was used because there are multiple time series to be considered jointly. The and time series were both considered exogenous, so that hypothetical future values could be considered in comparison of release plans, as discussed in the Motivation section.

## Model Specification & Estimation

By selecting , a maximum model order is obtained by

Models of order were estimated for later diagnostic checking.

## Model Diagnostic Checking

Candidate models were tested for stability and inadequacy. A 5% significance level was used in the Ljung-Box test. The results for each windowed period are shown in Table 4. All model orders were stable for all windowed periods. Several model orders were found to be inadequate by the Ljung-Box test: orders 1-2 for period *W2-79*, and order 5 for period *W158−235*.

1. Results of running stability and Ljung-Box test on each windowed period.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Model order | *W2-79* | | *W80−157* | | *W158−235* | |
| Stable | p-value | Stable | p-value | Stable | p-value |
| 1 | Yes | 0.009061 | Yes | 0.4478 | Yes | 0.09453 |
| 2 | Yes | 0.01401 | Yes | 0.5866 | Yes | 0.1255 |
| 3 | Yes | 0.2052 | Yes | 0.6470 | Yes | 0.1753 |
| 4 | Yes | 0.1288 | Yes | 0.7596 | Yes | 0.09363 |
| 5 | Yes | 0.3363 | Yes | 0.6133 | Yes | 0.04656 |
| 6 | Yes | 0.2818 | Yes | 0.3838 | Yes | 0.05703 |

## Model Selection

Models that were not rejected for instability or inadequacy were then compared and the best for each windowed period was selected by AIC selection criterion. The results of selection are shown in Table 5, with orders 4, 1, and 1 being chosen for periods *W2-79*, *W80−157*, and *W158−235*, respectively. The fit for each of these models is demonstrated by plotting one-step predictions along with actual values, shown for each model in Fig. 7.

1. Results of model selection, using AIC score to compare models of different order.

|  |  |  |  |
| --- | --- | --- | --- |
| Model order | AIC score | | |
| W2-79 | W80−157 | W158−235 |
| 1 | N/A | 429.8 | 477.9 |
| 2 | N/A | 439.3 | 482.4 |
| 3 | 400.8 | 440.9 | 489.7 |
| 4 | 400.3 | 450.2 | 499.9 |
| 5 | 404.0 | 456.7 | N/A |
| 6 | 414.9 | 461.7 | 508.8 |

|  |
| --- |
|  |
|  |
|  |

1. One-step predictions vs actual values, for each model selected by AIC score.

#### Related Work

Software defect (bug) prediction typically involves a detailed analysis of code or proposed design changes. Some of these analytical methods are mentioned next. Then several statistical approaches to prediction are discussed.

## Code Analysis Approaches

Akiyama [1] predicted defect counts based on lines of code (LOC), number of decisions, and the number of subroutine calls. Gafney [6] likewise predicted defect count based on LOC. Rather than code itself, Henry and Kafura [9] define metrics that are based on information taken from design documents, to be used in defect prediction. Nagappan and Ball [13] use relative code churn (lines modified) as a metric for predicting the density of defects. Giger, Pinzger, and Gall [7] compare the use of code churn to a more fined-grained approach, capturing “the exact code changes and their semantics down to statement level.”

## Statistical Approaches

Rather than requiring a detailed code analysis to predict defects, the approach proposed in this paper is to develop a mathematical model based on historical data on defect occurrences. Specifically, the proposed approach is to develop a defect prediction model using previous software features, improvements, and defects.

A related approach, used by Li, Shaw, Herbsleb, Ray, and Santhanam [11], is to study only the defect occurrences themselves, and attempt to develop a mathematical model for defect projection. In their work, functions were fitted to a time series of defect occurrences, then the function parameters themselves were extrapolated for each new release. They found that the Weibull model fit best in 73% of the tested software releases. They attempted to extrapolate model parameters using naive methods, moving averages, and exponential smoothing, but found these techniques to be “...inadequate in extrapolating model parameters of the Weibull model for defect-occurrence projection”. The reason given for this ineffectiveness is the changing nature of the software development system. For example, development practices, staffing levels, and usage patterns may all change between releases.

In another related approach, Graves, Karr, Marron, and Siy [8] developed several models that predict the future distribution of software faults in a given code module. The basis of their predictive models is a statistical analysis of change management data, which describes only the changes made to code files. The best model they found was a weighted time damping model, where every change in the module files contributed to fault prediction, with time-damping to account for age of changes. They achieved “slightly less successful performance” by basing a generalized linear model on just the modules age and the number of past changes. They also found factors that did not improve model performance: module length, number of developers making changes in the module, and how often a module is changed simultaneously with another module.

In the final approach discussed here, by Singh, Abbas, Ahmad, and Ramaswamy [14], the Box-Jenkins method is applied to datasets from the Eclipse and Mozilla software projects, which are represented as time series data, and defect count is predicted using an ARIMA model. Their modeling effort is focused at the component-level, and they conclude that “current bug count of a component is linearly related to its previous bug count”.

#### Conclusions and Future Work

##### Acknowledgment

##### References

1. F. Akiyama. An example of software system debugging. In IFIP Congress (1), volume 71, pages 353–359, 1971.
2. A. J. Bagnall, V. J. Rayward-Smith, and I. M. Whittley. The next release problem. Information and software technology, 43(14):883–890, 2001.
3. S. Bisgaard and M. Kulahci. Time series analysis and forecasting by example. John Wiley & Sons, 2011.
4. G. E. P. Box, G. M. Jenkins, and G. C. Reinsel. Time Series Analysis. John Wiley, 2008.
5. P. H. Franses. Time series models for business and economic forecasting. Cambridge university press, 1998.
6. J. E. Gaffney. Estimating the number of faults in code. Software Engineering, IEEE Transactions on, SE-10(4):459–464, July 1984.
7. E. Giger, M. Pinzger, and H. C. Gall. Comparing fine-grained source code changes and code churn for bug prediction. In Proceedings of the 8th Working Conference on Mining Software Repositories, pages 83–92. ACM, 2011.
8. T. L. Graves, A. F. Karr, J. S. Marron, and H. Siy. Predicting fault incidence using software change history. Software Engineering, IEEE Transactions on, 26(7):653–661, 2000.
9. S. Henry and D. Kafura. The evaluation of software systems’ structure using quantitative software metrics. Software: Practice and Experience, 14(6):561–573, 1984.
10. H. Jiang, J. Zhang, J. Xuan, Z. Ren, and Y. Hu. A hybrid ACO algorithm for the next release problem. In Software Engineering and Data Mining (SEDM), 2010 2nd International Conference on, pages 166–171. IEEE, 2010.
11. P. L. Li, M. Shaw, J. Herbsleb, B. Ray, and P. Santhanam. Empirical evaluation of defect projection models for widely-deployed production software systems. SIGSOFT Softw. Eng. Notes, 29(6):263–272, Oct. 2004.
12. T. K. Moon and W. C. Stirling. Mathematical methods and algorithms for signal processing, volume 1. Prentice hall New York, 2000.
13. N. Nagappan and T. Ball. Use of relative code churn measures to predict system defect density. In Software Engineering, 2005. ICSE 2005. Proceedings. 27th International Conference on, pages 284–292. IEEE, 2005.
14. L. L. Singh, A. M. Abbas, F. Ahmad, and S. Ramaswamy. Predicting software bugs using arima model. In Proceedings of the 48th Annual Southeast Regional Conference, page 27. ACM, 2010.
15. J. Xuan, H. Jiang, Z. Ren, and Z. Luo. Solving the large scale next release problem with a backbone-based multilevel algorithm. Software Engineering, IEEE Transactions on, 38(5):1195–1212, 2012.
16. K. Yang and C. Shahabi. On the stationarity of multivariate time series for correlation-based data analysis. In Data Mining, Fifth IEEE International Conference on, pages 4–pp. IEEE, 2005.
17. Y. Zhang, M. Harman, and S. A. Mansouri. The multi-objective next release problem. In Proceedings of the 9th annual conference on Genetic and evolutionary computation, pages 1129–1137. ACM, 2007.

1. *MongoDB* is a document-oriented, NoSQL database, and is available under the [GNU Affero GPL](https://gnu.org/licenses/agpl.html) license. [↑](#footnote-ref-1)
2. The *dse* library for R provides tools for time series models [↑](#footnote-ref-6)
3. The *stats* library for R provides core statistics functions. [↑](#footnote-ref-7)