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. This paper presents a time series model that uses historical information to predict the number of defects for the next product release based on hypothetical values for features and improvements completed in the next release, as well as historical defect information. This allows hypothetical release plans to be compared to assess their predicted impact on testing and defect-fixing time. We found that …

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 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 and effort required for testing and bug-fixing will likely be a function of the defects introduced during development, it is desirable to be able to predict the defects that can be expected as development proceeds.

A potential application for a defect prediction model is for comparing different release plans according to their estimated bug fallout. This would assist release planners 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, which is the focus of The Next Release Problem [2], a key problem in Search-Based Software Engineering (SBSE) [10, 15, 17].

Most previous approaches to defect prediction focus on either code analysis [**add refs**] or historical defect information [**add refs**]. However, for the defect prediction model to be useful in comparing release plans, the model should also depend on the basic elements of the release plan: the features and improvements planned for the next release, and the defects from past releases.

This paper presents an approach to defect prediction that predicts defects for a proposed release. This approach uses a multivariate time series model that incorporates information about proposed features and improvements, as well as historical defect data.

This paper proceeds as follows. First, further motivation for the use of a time-series model is presented in Section . Next, we present some background about time series modelling in Section . Section IV presents our modelling methodology and Section V present the application the time-series modelling approach to data from the *MongoDB*[[1]](#footnote-1) software project. Related work is presented in Section VI, and the paper concludes in Section .

#### Motivation

A use case for defect prediction is in planning software releases. The predicted number of defects can be used to estimate the time that will be required for testing and bug fixing for a planned release.

Release planners typically rely on their experience and project conventions to generate a release plan. A common approach is to use an ad-hoc method where one selects a set of features and improvements such that the estimated time to fix and test defects will not exceed the budgeted development time. However, if the estimation technique for the impact of defects is based on past experience, this can lead to significant inaccuracies. For example, a release planner could use the approach whereby if a previous release had 10 features and improvements which led to 2 weeks of bug fixing, then if the next release has 5 features and improvements, the time to fix bugs will either be the same amount of time (2 weeks) or a prorated amount (1 week).

A problem with this approach is that the estimation of bug fixing time is course. If you were to try and compare release plans with different subsets of features and improvements, the course nature of the estimation technique would be hard to distinguish the impact of the features and improvements ont he bug fixing time.

The alternative is to have a model-based approach where the model must take into account the differences in composition of features and improvements between the 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. Such a model would assume some explanatory relationship as shown in Fig. 1.

The use of such a model may give release planners a more accurate means for evaluating the additional development time needed to address bug fallout for a given hypothetical release plan. By improving the accuracy of defect prediction, the release planner can ensure sufficient time in the schedule to fix bugs, thereby maintaining a high software quality and giving the release planner the freedom to maximize the expected value or revenue of the next release.



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

A planner could then proceed to optimize the subset of requirements planned for the next release. This is the goal of the Next Release Problem [2]. 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. 2.



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, time series and autoregressive models are introduced. Then, further concepts related to modeling, exogeneity and stationarity, are 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. This dependence is termed autocorrelation, and accounting for it is one of the primary functions of a time series model.

## Autoregressive Models

A basic autoregressive (AR) model is formed as a linear combination of previous values, plus a white noise term that account for random variations (the stochastic portion). An model for predicting value at time can be written

,

where are the parameters, is a constant, and is the white noise term.

## Vector AR Models

When the AR model is extended to the multivariate case (i.e. allowing for multiple time series), a Vector AR (VAR) model is formed. This model will support not only a time series for defect count, but also time series for the two release plan variables: improvements and new features.

## Endogeneity and Exogeneity

Under the VAR model, the behavior of each time series is explained by its own past values and the past values of the other time series. Each variable then would be called endogenous.

The alternative is that a time series should not be explained itself, and is only used to explain other time series. This type of explanatory variable is called exogenous, and could be considered an input.

By also considering exogenous variables, a VAR model would become a VARX model. This model meets the requirements of the explanatory model described in the Motivation section, since it would allow release plan variables to be kept exogenous and used only to explain defect count.

## Trends

AR, VAR, and VARX models do not account for non-stationary data. If a time series is not stationary, differencing may produce a stationary series. Trends and tests for stationarity will be discussed next.

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 variations occur, and the series 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.

#### 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.

## 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.

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. The Ljung-Box test is used for this purpose.

## 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, including Akaike Information Criterion (AIC), AIC with correction (AICc), and Bayesian Information Criterion (BIC). Bisgaard and Kulahci noted that “[t]he penalty for introducing unnecessary parameters is more severe for BIC and AICC than for AIC” [3]. A less severe penalty for the number of parameters would be preferred in this case, since we are already limiting the number of parameters in the model specification step, and because additional parameters may in fact be necessary to account for time series autocorrelations with higher lags. Therefore, AIC was chosen as the selection criterion.

#### Application of Methodology

To validate our approach of using a time-series model to predict defects, we used historical data taken from a software project’s issue tracking system. Issue tracking systems are used by projects for tracking development tasks, features, enhancements, and bugs, both past and present.

We chose the *MongoDB* Core Server project as the data set. This project was chosen as it has been active since May 2009 and uses *JIRA*[[2]](#footnote-2) for issue tracking, which made it easy to collect data. Issues for versions 0.9.3 through 3.0.0-rc6 were exported from the project’s *JIRA* web interface into XML format. The fields collected from each issue report was: type, priority, creation date, and resolution date.

As the proposed model structure assumes that bug creation can be explained by software changes, issues that do not result in any change should not be included in the dataset. For this reason, only issues with resolution *fixed*, *complete*, or *done* were kept. In the data collected, 18 (0.26%) issues did not meet this criterion and were excluded. Also, *JIRA* supports issues having sub-tasks. We treated sub-tasks the same as issues, and converted them to be the same type as their parent issue. Those sub-tasks whose parent issue was not in the dataset were considered orphans and discarded. There were 20 (0.28%) orphaned sub-tasks in the dataset. The final dataset contained 7042 issues.

## Data Preparation

After creation, the dataset was operated on to prepare it for time series modeling. The data was sampled, made stationary, and windowed. These three steps are discussed next.

## Sampling

First, the data was sampled at regular periods to measure the following: number of improvements resolved, number of features resolved, and number of bugs created. A 7-day sampling period was used.This sampling process is illustrated in Fig. 3 and results in Table 1.

## Establishing Stationarity

To establish stationarity, the ADF unit root and KPSS stationarity tests were applied. In both tests, it was assumed that the deterministic component was constant (without slope). The result of the tests are listed in Table 1.

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

As the result of the unit root and stationarity test (Table 2) agreed, we rejected the hypothesis that a unit root (stochastic trend) is present at the 1% significance level and we failed to reject the hypothesis of stationarity with greater than 10% significance. Hence, the differenced time series (see Fig. 3) were used for modeling, and are referred to as , , and .

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. 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

It can be assumed that the software development process underlying a given project changes over time. Rather than developing a model that also changes over time, the data was kept for modeling only if it occurred within a time window. This was done to limit the effect of process change on the model. A time window of 78 weeks (approximately 18 months) was selected to balance between more observations (to capture consistent long-term behaviors), and fewer observations (to limit exposure to behavioral changes).

Applying this time window, the data was divided into three 78-week windows. As the data was differenced, the first sample was skipped in each data period. These windowed periods are denoted *W2-79*, *W80−157*, and *W158−235*.

## VARX Modeling

## Use of the VARX Model

As discussed in Section III, the model was chosen to model the time series 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 when comparing hypothetical release plans.

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, specifically 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 was demonstrated by plotting one-step predictions along with actual values, as 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.

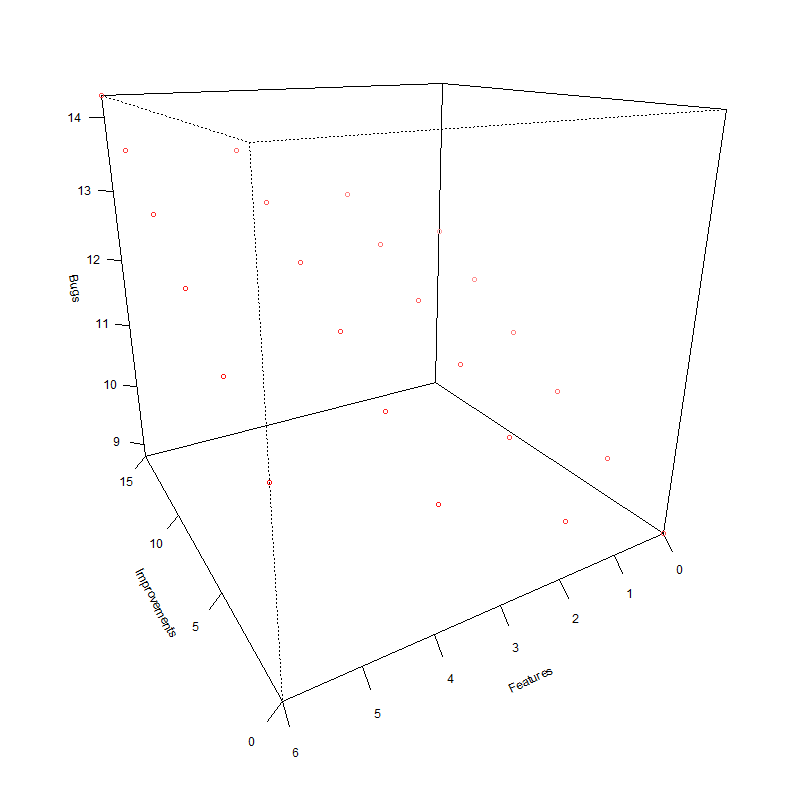
#### Forecasting

The model selected for each windowed period was then applied to forecast a future output value just past the end of the window. The desired interface for making these predictions is to provide as input the number of improvements and features that are expected to be resolved, and to receive as output the predicted number of bugs that will be created. In order to provide this interface, input values will be converted to differences, so prediction can be performed. Then differencing will be removed from the resulting prediction. This will hide the fact that the underlying model is operating with differenced time series data.

The resulting prediction data is shown only for the first time window, *W2-79*, and is listed in Table 5, including upper and lower bounds from confidence intervals. A 3D visualization of only the mean predicted values is in Fig. 5.

1. Forecasting at the end of the first time window, *W2-79.* future output values are predicted for a number of hypothetical input values.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Improvements | Features | 90% lo | 75% lo | mean | 75% hi | 90% hi |
| 0 | 0 | 4.42 | 5.73 | 8.79 | 11.85 | 13.16 |
| 0 | 2 | 5.26 | 6.57 | 9.63 | 12.69 | 14.01 |
| 0 | 4 | 6.10 | 7.42 | 10.47 | 13.53 | 14.85 |
| 0 | 6 | 6.94 | 8.26 | 11.32 | 14.38 | 15.69 |
| 3 | 0 | 5.01 | 6.32 | 9.38 | 12.44 | 13.76 |
| 3 | 2 | 5.85 | 7.17 | 10.22 | 13.28 | 14.60 |
| 3 | 4 | 6.69 | 8.01 | 11.07 | 14.13 | 15.44 |
| 3 | 6 | 7.54 | 8.85 | 11.91 | 14.97 | 16.28 |
| 6 | 0 | 5.60 | 6.92 | 9.97 | 13.03 | 14.35 |
| 6 | 2 | 6.44 | 7.76 | 10.82 | 13.88 | 15.19 |
| 6 | 4 | 7.29 | 8.60 | 11.66 | 14.72 | 16.03 |
| 6 | 6 | 8.13 | 9.44 | 12.50 | 15.56 | 16.88 |
| 9 | 0 | 6.19 | 7.51 | 10.57 | 13.63 | 14.94 |
| 9 | 2 | 7.04 | 8.35 | 11.41 | 14.47 | 15.78 |
| 9 | 4 | 7.88 | 9.19 | 12.25 | 15.31 | 16.63 |
| 9 | 6 | 8.72 | 10.04 | 13.10 | 16.15 | 17.47 |
| 12 | 0 | 6.79 | 8.10 | 11.16 | 14.22 | 15.53 |
| 12 | 2 | 7.63 | 8.94 | 12.00 | 15.06 | 16.38 |
| 12 | 4 | 8.47 | 9.79 | 12.84 | 15.90 | 17.22 |
| 12 | 6 | 9.31 | 10.63 | 13.69 | 16.75 | 18.06 |
| 15 | 0 | 7.38 | 8.69 | 11.75 | 14.81 | 16.13 |
| 15 | 2 | 8.22 | 9.54 | 12.59 | 15.65 | 16.97 |
| 15 | 4 | 9.06 | 10.38 | 13.44 | 16.50 | 17.81 |
| 15 | 6 | 9.91 | 11.22 | 14.28 | 17.34 | 18.65 |



1. Forecasting at the end of the first time window, *W2-79.* future output values are predicted for a number of hypothetical input values. Only mean predicted value is shown.

The predicted output value follows an increasing relationship with input values, with features having a stronger effect than improvements. Data from the two other time windows are not shown, but it should be known that the other results are similar for the most part. However, the predictions for the second time window showed a slight decreasing relationship between the number of features and the predicted number of bugs.

#### Related Work

Prior defect prediction techniques generally fall into two categories; those based on code analysis and those based on statistical analysis. Code analysis techniques typically involves a detailed analysis of code or proposed design changes using metrics such as lines of code (LOC) or decision points. Statistical analysis techniques create mathematical models based on historical defect occurrence information. This section presents an overview of some of the previous work on defect prediction that fall into these two categories.

## 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] defined metrics that were based on information taken from design documents, to be used in defect prediction. Nagappan and Ball [13] used relative code churn (lines modified) as a metric for predicting the density of defects. Giger, Pinzger, and Gall [7] compared 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 of 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], was 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. Unlike their approach, we consider features and improvements in addition to defects, and address the changing nature of software development practice by the use of time windows.

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.

Finally, Singh, Abbas, Ahmad, and Ramaswamy [14], applied the Box-Jenkins method to time series datasets from the Eclipse and Mozilla software projects to predict defect counts using an ARIMA model. Their modeling effort was focused at the component-level, and they concluded that “current bug count of a component is linearly related to its previous bug count”.

#### Conclusions and Future Work

The VARX modeling methodology was successfully applied to time series data collected from the *MongoDB* project. A model was created for each of three time windows, and then used to make defect predictions for a range of hypothetical values for improvements and features. Data from these predictions tended to show an increasing relationship with the input values. Of concern still is the slight decreasing relationship found between features and bugs for the predicted values after the second time window.

Now that the methodology has been applied to one project dataset, it remains to be seen how well it will work for others. More commonly studied software projects, like *Eclipse* or *Mozilla* might be good choices.

##### 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. See https://www.mongodb.org/. [↑](#footnote-ref-1)
2. *JIRA* is an issue tracking and project management system made by Atlassian, who provide free JIRA subscriptions for qualified open source projects. [↑](#footnote-ref-2)