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. A time series model is presented that uses historical project information to predict the number of future defects, given the number of future features and improvements completed. This would allow hypothetical release plans to be compared by assessing their predicted impact on testing and defect-fixing time. We selected the VARX time series model as a reasonable approach. The accuracy of the model appeared low for a single dataset, but the error was found to be normally distributed.**

Keywords-software defect prediction; quality assurance; release planning; time series model;

I. 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 to accommodate fixing 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 in 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 number of expected defects.

A potential application for defect prediction is to compare different release plans according to their estimated bug fallout and subsequent impact on testing and bug-fixing times. 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 approaches to defect prediction focus on either code analysis [1, 6, 7, 9, 13] or historical defect information [8, 11, 14]. However, for the defect prediction model to be useful in comparing release plans, the model should also depend on the planned features and improvements planned for the next release, as well as the defects from past releases.

This paper presents an approach to defect prediction that can be applied for a proposed release. A multivariate time series model is used that incorporates information about proposed features, improvements, and historical defect data.

The paper proceeds as follows. First, Section II presents further motivation for the use of a time series model for predicting defects. Next, we present an overview of concepts in time series modeling in Section III. Section IV presents our modeling methodology and Section V presents the application of the approach, which is applied to a software project dataset. Related work is presented in Section VI, and the paper is concludes in Section VII.

II. MOTIVATION

Release planners typically rely on both their experience and project conventions to generate a release plan by selecting planned features and improvements such that the estimated time to test for and fix defects will not cause a schedule slip.

However, if the defect estimation technique is only loosely based on past experience, as with a rule-of-thumb, then it may prove too coarse for comparing multiple release plans. Specifically, such a technique may not provide any quantitative difference between release plans that are similar (but not the same). For example, suppose two different release plans are being considered. Both include two features, but one has five improvements and the other has seven. A rule-of-thumb approach may provide the same estimate for each. Even for dissimilar release plans, such an approach still has the disadvantage of lacking confidence intervals to quantify prediction uncertainty.

An alternative approach is to develop a model that will 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 and that prediction uncertainty can be quantified by confidence intervals. Such a model would assume some explanatory relationship, such 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 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 a means to compare any number of release plans.

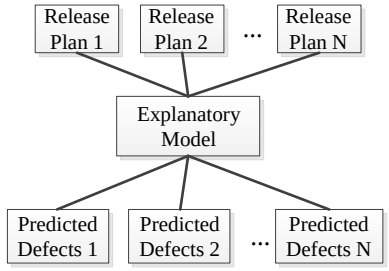

Figure 1. Using an explanatory model allows for the possibility of different defect predictions for each release plan.

Since predictive models rarely have perfect accuracy, confidence levels are an important part of any prediction. Accounting for the confidence of a prediction allows release planners to assess the risk of relying on the defect prediction. Planners can choose a more narrow 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.

III. Time Series Modeling

In this section, time series and autoregressive models are introduced. Then, further concepts related to modeling, exogeneity and stationarity, are discussed.

A. 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.

B. Autoregressive Models

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

$$X_t = c + \sum_{i=1}^{p} \phi_t X_{t-1} + \epsilon_t$$

where ϕ_1, ϕ_2, …, ϕ_p are the p parameters, c is a constant, and ε_t is the white noise term.

C. 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.

D. Endogeneity and Exogeneity

Under the VAR model, the behavior of each time series is explained by both its own past values and the past values of the other time series. This makes the variables “endogenous".

The alternative is that a time series should not be explained by 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.

E. 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 vary over time, and are therefore not interpretable [5]. Two trend types are discussed here: deterministic and stochastic.

A deterministic trend will move 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 to assess if a deterministic or stochastic trend is present.

F. 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 established by assuming normality [4].

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

G. 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 the 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.

IV. 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.

A. Model Specification & Estimation

The specification of a *VARX(p)* model is accomplished by choosing an order *p*, which is the number of autoregressive terms to include in the model. Then 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 is 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 *n* be the number of time samples in a time series. When there are *m* time series, each sample contains *m* observations, so there are *mn* total observations for all time series. Next, for a *VARX(p)* model of the m time series variables, there are m^2p unknown parameters to be estimated. Let the ratio of observations to parameters be denoted by

$$K = \frac{mn}{m^2 p} = \frac{n}{mp}$$

To keep *K* at or above some minimum ratio K_{min}, we form the inequality

$$K_{min} \leq K = \frac{n}{mp}$$

In terms of *p* this becomes

$$p \leq \frac{n}{mK_{min}}$$

For a fixed value of K_{min}, an upper bound on the model order would be

$$p_{max} = \left\lfloor \frac{n}{mK_{min}} \right\rfloor$$

With this upper bound, model specification will include the generation of models having order 1, 2, …, p_{max}. These models, with their estimated parameters, will be candidates for final model selection after undergoing diagnostic checking.

B. 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 Autoregressive-moving averages (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.

C. 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.

V. 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*[1] Core Server project as the data set. This project was chosen as it has been active since May 2009 and uses *JIRA*[2] for issue tracking, which made it easy to

[1] MongoDB is an open source, document-oriented database system.

[2] JIRA is an issue tracking and project management system made by Atlassian.

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 were: type, priority, creation date, and resolution date.

As the proposed model structure assumes that bug creation can be explained by software changes, issues not resulting in a change should not be included in the dataset. For this reason, only *fixed*, *complete*, or *done* issues 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.

A. 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.

1) 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.

2) 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 results of the tests are listed in Table I.

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 new results of the unit root and stationarity test (Table II) 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. 2) were used for modeling ($Y_{\Delta bug}$, $Y_{\Delta imp}$, and $Y_{\Delta new}$).

TABLE I. RESULTS OF RUNNING THE ADF UNIT ROOT TEST AND KPSS STATIONARITY TEST ON Y_{BUG}, Y_{IMP}, AND Y_{NEW}.

Statistic	$\mathbf{Y_{bug}}$		$\mathbf{Y_{imp}}$		$\mathbf{Y_{new}}$	
	Value	***p-value***	***Value***	***p-value***	***Value***	***p-value***
ADF (τ_2)	-5.020	< 1%	-7.402	< 1%	-7.845	< 1%
ADF (ϕ_2)	12.65	< 1%	27.42	< 1%	30.77	< 1%
KPSS	2.852	< 1%	2.021	< 1%	0.5269	2.5-5%

TABLE II. RESULTS OF RUNNING THE ADF UNIT ROOT TEST AND KPSS STATIONARITY TEST ON $Y_{\Delta BUG}$, $Y_{\Delta IMP}$, AND $Y_{\Delta NEW}$ (SECOND RUN).

Statistic	$\mathbf{Y_{\Delta bug}}$		$\mathbf{Y_{\Delta imp}}$		$\mathbf{Y_{\Delta new}}$	
	Value	***p-value***	***Value***		***Value***	***p-value***
ADF (τ_2)	-17.65	< 1%	-20.44	< 1%	-21.90	< 1%
ADF (ϕ_2)	155.8	< 1%	208.9	< 1%	239.8	< 1%
KPSS	0.0115	> 10%	0.0127	> 10%	0.0127	> 10%

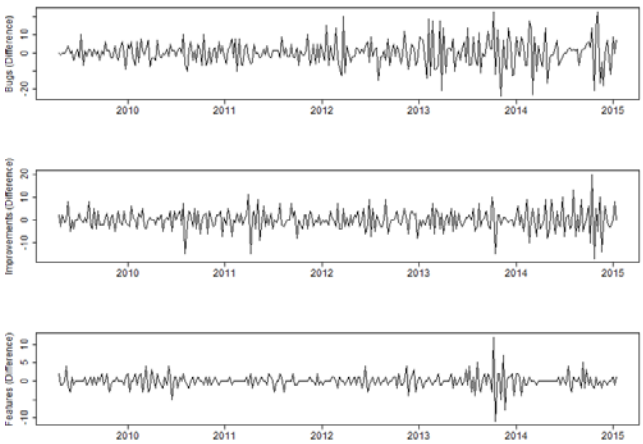

Figure 2. Differenced time series data.

3) 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 $W_{2\text{-}79}$, $W_{80\text{–}157}$, and $W_{158\text{–}235}$.

B. VARX Modeling

1) Use of the VARX Model

As discussed in Section III, the *VARX* model was chosen to model the time series because there are multiple time series to be considered jointly. The $Y_{\Delta imp}$ and $Y_{\Delta new}$ time series were both considered exogenous, so that hypothetical future values could be considered when comparing release plans.

By selecting K_{min}=4, a maximum model order is obtained by

$$p_{max} = \left\lfloor \frac{78}{(3)(4)} \right\rfloor = \lfloor 6.5 \rfloor = 6$$

So models of order 1 through p_{max} = 6 were estimated for later diagnostic checking.

C. 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 III. All model orders were stable for all windowed periods. Several model orders were found to be inadequate, specifically orders 1-2 for period $W_{2\text{-}79}$, and order 5 for period $W_{158\text{–}235}$.

TABLE III. RESULTS OF RUNNING STABILITY AND LJUNG-BOX TEST ON EACH WINDOWED PERIOD.

Model order	W_{2-79}		W_{80-157}		$W_{158-235}$	
	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

D. 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 the bolded values shown in Table IV. 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. 3. The fit for each appears to track well with many of the significant changes in the time series.

TABLE IV. RESULTS OF MODEL SELECTION, USING AIC SCORE TO COMPARE MODELS OF DIFFERENT ORDER.

Model order	AIC score		
	W_{2-79}	W_{80-157}	$W_{158-235}$
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

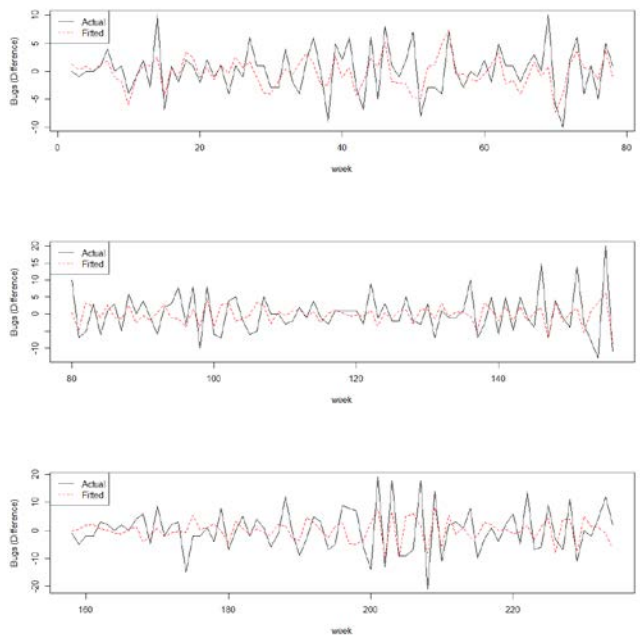

Figure 3. One-step predictions vs actual values, for each model selected by AIC score.

E. Forecasting

The model selected for each windowed period (model orders 4, 1 and 1) was used to forecast the number of defects in the next sample after the end of the window. The input for making these predictions was the number of improvements and features that were expected to be resolved. The input values were converted to differences, since the underlying model was formed using differenced data. Differencing was then removed to provide the predicted number of future defects.

Table V shows the resulting single-step, out-of-sample defect prediction data for the first time window, W_{2-79}, including the upper and lower bounds of the confidence intervals. The actual number of improvements, features, and bugs in the prediction sample period was 4, 0, and 18, respectively. Notice that the actual number of bugs, 18, is outside of the 90% confidence interval, which spans from 6.4 to 13.79 (see the outlined row in Table 5). On the other hand, the actual number of future defects in the next window, W_{80-157}, was 17. This was inside the 90% confidence interval, which spans from 13.38 to 18.00.

TABLE V. FORECASTING AT THE END OF THE FIRST TIME WINDOW, W_{2-79}. FUTURE OUTPUT VALUES ARE PREDICTED FOR A NUMBER OF HYPOTHETICAL INPUT VALUES.

Improvements	Features	90% lo	75% lo	mean	75% hi	90% hi
2	0	5.61	6.72	9.31	11.89	13.00
2	1	5.54	6.66	9.24	11.82	12.93
2	2	5.48	6.59	9.17	11.75	12.86
2	3	5.41	6.52	9.10	11.69	12.80
4	0	6.40	7.51	10.09	12.68	13.79
4	1	6.33	7.44	10.03	12.61	13.72
4	2	6.27	7.38	9.96	12.54	13.65
4	3	6.20	7.31	9.89	12.48	13.59

To gauge how well prediction will work in general, a sliding 78-week window was applied. The sliding window started at the first sample period, and was shifted by one sample period after modeling. Only the actual number of improvements and features were used in this forecasting. The resulting distribution of errors between the mean forecasted bugs and the actual number of bugs is shown as a histogram in Fig. 4. Note that the histogram appears to be normally distributed. The actual number of bugs was inside the 90% confidence interval for 23.87% of the sliding window ranges.

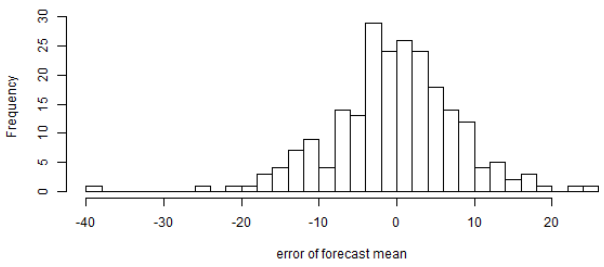

Figure 4. Histogram of forecast mean errors obtained using a 78-week sliding window.

VI. 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 involve 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.

A. Code Analysis Approaches

Akiyama [1] and Gafney [6] predicted defect counts based on lines of code (LOC), number of decisions, and the number of subroutine calls. Rather than code itself, Henry and Kafura [9] defined metrics from design document information for use in defect prediction. Both Nagappan and Ball [13] and Giger, Pinzger, and Gall [7] used relative code churn as a metric for predicting the density of defects.

B. Statistical Approaches

Li et al. [11] studied defect occurrences to develop a mathematical model for defect projection that is based only on past defect occurrences. In their work, functions were fitted to a time series of defect occurrences, and then the function parameters were extrapolated for each new release. A 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 due to changes in development practices, staffing levels, and usage patterns between releases. We use time windowed data to limit the effects from changing software development practice.

Graves et al. [8] developed several models to predict the future distribution of software faults in a given code module. Their predictive models used a statistical analysis of change management data, which describes only the changes made to code files. They found the best model 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. The model we use is instead based on project issue tracking data, and includes changes to whatever modules are found in that project.

Finally, Singh et al. [14], applied the Box-Jenkins method to time series datasets from the Eclipse and Mozilla projects to predict defect counts using an ARIMA model. Their modeling effort was focused at the component-level, and found a linear relationship between the current bug count of a component and its previous bug count. Their model is only in terms of past defects. We include past features and improvements as model inputs, so defects can be predicted using values from hypothetical release plans.

VII. Conclusions and Future Work

The VARX modeling methodology was successfully applied to the 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 the number of improvements and features. Also, a picture of the prediction performance was obtained by applying the approach with a sliding window. This resulted in a normally distributed error between the mean forecasted and actual number of bugs. A low proportion (23.87%) of the sliding window ranges included the actual number of bugs using a 90% confidence interval. These results indicate that the VARX model had a low prediction accuracy for the actual number of defects in the *MongoDB* dataset.

Having applied the VARX time series model to one project dataset, a next step is to apply the methodology to other software project data sets, such as *Eclipse* or *Firefox*, to better determine the applicability of the modeling approach.

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.