Using Time Series Models for Defect Prediction in Software Release Planning

James Tunnell

March 27, 2015

Abstract

To produce a high-quality software release, sufficient time should be allowed for testing and fixing defects. Otherwise, there is a risk of slip in the development schedule and/or software quality. A time series model is used to predict the number of bugs created during development. The model depends on the previous numbers of bugs created. The model also depends, in an exogenous manner, on the previous numbers of new features resolved and improvements resolved. This model structure would allow hypothetical release plans to be compared by assessing their predicted impact on testing and defect-fixing time. The VARX time series model was selected as a reasonable approach. The accuracy of the model appeared low for a single dataset, but the error was found to be normally distributed.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Two primary concerns in software release planning are: improving functionality and maintaining quality. Both objectives are constrained by limits on development time and cost. In order to respect these constraints and still pursue both objectives, the scope of planned work must be limited so that time is available to properly deal with the inevitable defects (bugs) that will arise. In this way, a software release can better ensure quality while also improving functionality.

A critical step in this planning process is to factor in a suitable amount of time for testing and bug-fixing. Otherwise, there is a risk of slip in the development schedule and/or software quality. As the time and effort 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.

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 projects 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 (discussed in detail in the Motivation section).

Many approaches to defect prediction focus on either code analysis or historical defect information. To make the defect prediction model useful for comparing release plans, the model must be dependent in some way on the basic elements of the release plan: planned new features and improvements. The historical defect models discussed in the Related Work section are limited in this respect, as they depend only on the past defects.

An approach to defect prediction is presented using a multivariate time series model. This model can be applied for a proposed release, because predictions can be made using only information about proposed features and improvements.

The paper is organized as follows. First, related work is presented in Section 2. Then, Further motivation for the use of a time series model for predicting defects is presented in Section 3. Next, an overview of time series modeling concepts is provided in Section 4. Methodologies for data collection and time series modeling are detailed in Sections 5 and 6, respectively. The results of applying these methodologies are given in Section 7. Last, the paper concludes and poses future work in Section 8.

Chapter 2

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.

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 to Defect Prediction

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".

Chapter 3

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 Figure 3.1.

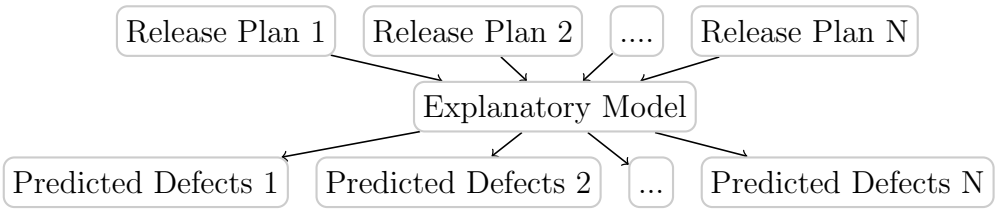


Figure 3.1: Using an explanatory model allows for the possibility of different defect predictions for each release plan.

Since predictive models will have some inaccuracy, but confidence levels can be used to quantify the uncertainty of future prediction based on past accuracy. This would allow release planners to assess the risk of relying on the defect prediction. A higher confidence level results in less risk because it encompasses a larger window for the prediction. Conversely, a lower confidence level results in more risk and a more narrow prediction window.

Application to the Next Release Problem

Release plan optimization is exactly the goal of The Next Release Problem (NRP), but there is a gap between the abstract domain of the NRP and the detailed, messy data found in software projects. By applying an explanatory predictive model there is a path toward bridging this gap, opening up the potential for using NRP optimization techniques in real-world release planning. In this section, first the NRP is described, then the gap between it and practical planning is discussed, and finally it is shown how the explanatory model suggested earlier would be applied to help bridge this gap.

Defining the NRP

The Next Release Problem (NRP) was defined by Bagnall, Rayward-Smith, and Whittley [2], and was shown to be NP-Hard. Being abstract in its treatment of feature cost, a broad range of optimization techniques can be applied to the NRP, such as integer programming, hill climbing, simulated annealing, genetic algorithms, etc. The NRP is the subject of academic research in the area of Search-Based Software Engineering [10, 15, 17].

The NRP describes the situation where software project planners, who have multiple customers to satisfy, would like to maximize the revenue produced from completing the project. This is all described mathematically as follows.

A software project has a set R of all possible requirements (new features and enhancements) that might be included in the next software release. A customer i is satisfied by completing a subset $R_i \subseteq R$. The importance of a customer i is given by the weight, $w_i \in \mathbb{Z}^+$.

Requirements may have acyclic dependencies, or prerequisites, that must be completed first. A subset that includes all prerequisite requirements, recursively, is indicated by $\hat{R}_i$, and should be taken to mean

$$\hat{R}_i = R_i \cup ancestors(R_i) \tag{3.1}$$

For example, if $R_1 = \{r_2\}$, and r_1 is a prerequisite for r_2, then $\hat{R}_1 = \{r_1, r_2\}$.

A requirement $r \in R$ has a cost $cost(r) \in \mathbb{Z}^+$, associated with its implementation, not considering the cost of any prerequisite requirements. Then, the cost for some subset $R' \subseteq R$ will be

$$cost(R') = \sum \{cost(r) | r \in \hat{R}_i\} \tag{3.2}$$

Once customer i is satisfied, their weight w_i contributes to the total revenue

from the project, as in

$$\sum_{i \in S} w_i \tag{3.3}$$

So, the NRP is posed as follows: for a group of n customers, select the subset $S \subseteq \{1, 2, ..., n\}$ that maximizes total revenue, while keeping the total cost within some budget constraint B. This is given by

$$\begin{aligned} &maximize \quad \sum_{i \in S} w_i \\ &subject\ to \quad cost(\bigcup_{i \in S} \hat{R}_i) \leq B \end{aligned} \tag{3.4}$$

The Gap Between Abstraction and Reality

As was discussed in the previous section, a planner would need several things to be able to implement a NRP-like optimization:

1. A set of requirements that could potentially be implemented.
2. A set of customers that are satisfied by some subset of the requirements, and have an associated weight.
3. A cost function, to quantify the cost of each requirement.
4. A cost budget that should not be exceeded.

Having all these in hand, a planner could proceed to optimize the subset of requirements planned for the next release. One difficulty with this that can be highlighted is in the definition of a cost function. It might be suggested that the estimated time to implement a requirement alone might be used to determine cost, but there is a practical detail that prevents this: in order to maintain quality software, the total cost of any requirement should take into consideration both the cost of implementation *and* the cost of fixing associated

defects. Otherwise, a release plan would appear to be within budget, when there is a risk that the budget will be exceeded when defect costs are also considered.

Bridging the Gap

We use the explanatory model to address the need to consider defect cost. Such a model, given some subset of proposed requirements, can be used to predict defects and to find additional cost which should be considered. This use of the predictive model is illustrated in Figure 3.2.

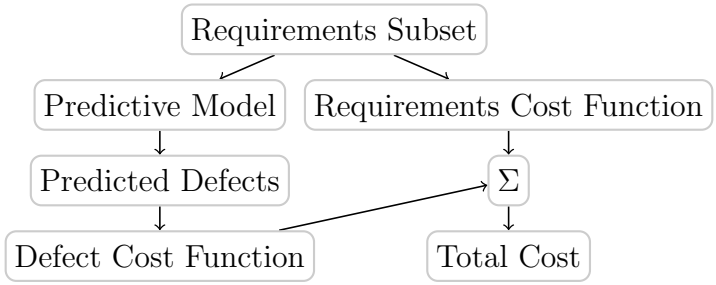


Figure 3.2: Defect prediction model used in determining overall cost of some requirements subset.

Since predictive models cannot be perfectly accurate, instead we would expect that any forecasting would include confidence levels. Taking into account the confidence of a prediction allows planners to account for risk in the use of the defect prediction. If more risk is acceptable, then planners will get a narrower prediction window, and in exchange take more of a chance that the prediction is inaccurate. A wider prediction window means, though, that when the defect prediction is used to determine requirements cost, that potential cost range will also be wider.

Chapter 4

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 accounts for random variations (the stochastic portion). An $AR(p)$ model for predicting a value X at time t can be

written

$$X_t = c + \sum_{i=1}^{p} \phi_t X_{t-1} + \epsilon_t \tag{4.1}$$

where $\phi_1, \phi_2, ..., \phi_p$ are the p parameters, c is a constant, and ϵ_t is the white noise term.

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 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.

Trends and Stationarity

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. 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.

Stationarity can be strict or weak (of some order). Strict stationarity occurs when statistical properties are invariant with respect to shifts of the time origin[12]. Alternatively, a weak stationarity (of second order) can be established, and from this 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], so the goal of stationarity testing will be to establish second-order stationarity for each univariate time series component, and then show that the assumption of normality is reasonable. This will establish the stationarity of the multivariate time series as a whole. Next, tests are discussed for assessing if a deterministic or stochastic trend is present.

Unit Root and Stationarity Testing

A time series that contains a stochastic trend is non-stationary. A pure autoregressive (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 below some significance,

the null hypothesis can be rejected, and it is established that the time series does not have a stochastic trend. 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 KwiatkowskiPhillipsSchmidtShin (KPSS) test can be applied for testing stationarity.

Chapter 5

Data Methodology

In this section, the data source and data collection method are detailed. Then, the method of preparing data for the modeling phase is presented.

Data Source

The empirical data used to establish a predictive model will be taken from software project historical data, found in an issue tracking system. In addition to tracking bugs, past and present, an issue tracking system can be used to track features, enhancements, or any other type of software process issue.

The *MongoDB* Core Server software project was selected for initial application of the modeling methodology. The project has been actively developed since 2009. Data from versions 0.9.3 through 3.0.0-rc6 are used. The dataset contained 7042 issues.

Data Collection & Cleansing

MongoDB uses JIRA[1] for issue tracking. Issue data is exported from the project's JIRA web interface as XML data. Then, issue data is extracted from the JIRA XML, and the following fields are kept from each issue: type, priority, creation date, resolution date.

Not all of the data was preserved for modeling. The modification or removal of data is discussed next.

Unfixed Issues

The proposed model structure assumes that bug creation can be explained by software changes. Therefore, 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* will be kept. Other possible issue resolutions are: *unresolved*, *won't fix*, *duplicate*, etc. In the data used, 18 (0.26%) of the issues were unfixed.

Sub-tasks

Issues that are sub-tasks are first converted to be the same type as the parent issue. Those sub-tasks whose parent issue is not in the dataset are considered orphans and discarded. There were 20 (0.28%) orphaned sub-tasks encountered in the dataset, so this decision is not expected to have much impact on the outcome.

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

[1] JIRA is an issue tracking and project management system made by Atlassian, who provide free JIRA subscriptions for qualified open source projects.

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. As an example, this sampling process is illustrated in Figure 5.1, with results shown in Table 5.1.

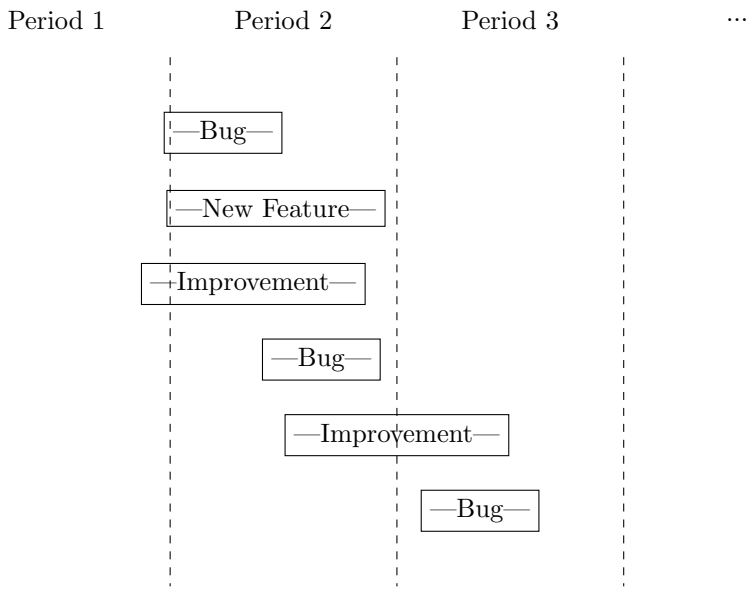


Figure 5.1: Sampling issue data by dividing time into equally-spaced periods.

Table 5.1: Results of sampling example issues shown in figure 5.1.

Period	Improvements Resolved	New Features Resolved	Bugs Created
1	0	0	1
2	1	1	1
3	1	0	1

Establishing Stationarity

To establish stationarity, we first need to see if we can rule out the presence of a stochastic trend by applying the augmented Dickey-Fuller (ADF) test. If we can indeed rule out a stochastic trend, we should be able to confirm stationarity by applying the KPSS test. Or, if a stochastic trend cannot be ruled out, then KPSS test should be applied to check that trend stationarity is also rejected. If the data is found to have a stochastic trend, it should be differenced and then retested to confirm (trend) stationarity. In both tests, it will be assumed that the deterministic component is constant, with an intercept but no trend.

Time Windowing

It is assumed that the software development process underlying a given project might change over time. Rather than developing a model that also changes over time, the data will be kept for modeling only if it occurs within a time window. This will limit the amount of process change the model is exposed to. Taking this approach means that the entire modeling methodology will be executed for each time-windowed part of the data.

Chapter 6

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 would be selection, where models are compared by some model selection criterion [4, pg. 581]. This section presents the approach used to specify, estimate, check, and select a VARX model to be used for defect prediction.

Model Specification & Estimation

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. 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 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

$$K = \frac{mn}{m^2p} = \frac{n}{mp} \tag{6.1}$$

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

$$K_{min} \leq K = \frac{n}{mp} \tag{6.2}$$

In terms of p this becomes

$$p \leq \frac{n}{mK_{min}} \tag{6.3}$$

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

$$p_{max} = \left\lfloor \frac{n}{mK_{min}} \right\rfloor \tag{6.4}$$

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.

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 model with an AR portion 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.

Portmanteau Test

For an adequate ARMA model, it can be shown that "As the series length increases, the [model residuals] become close to the white noise..." [4, p. 338]. For this reason, there are model inadequacy tests formed around a study of the residuals.

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 p-value should be above some level of significance, say 5%.

Model Selection

Model selection criteria are used to compare models according to their fit, by penalizing for residual error and 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.

Chapter 7

Results

Data Collection

The *MongoDB* dataset was collected according to the data methodology, 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 Y_{bug}, Y_{imp}, and Y_{new}, respectively, and are shown in figure 7.1.

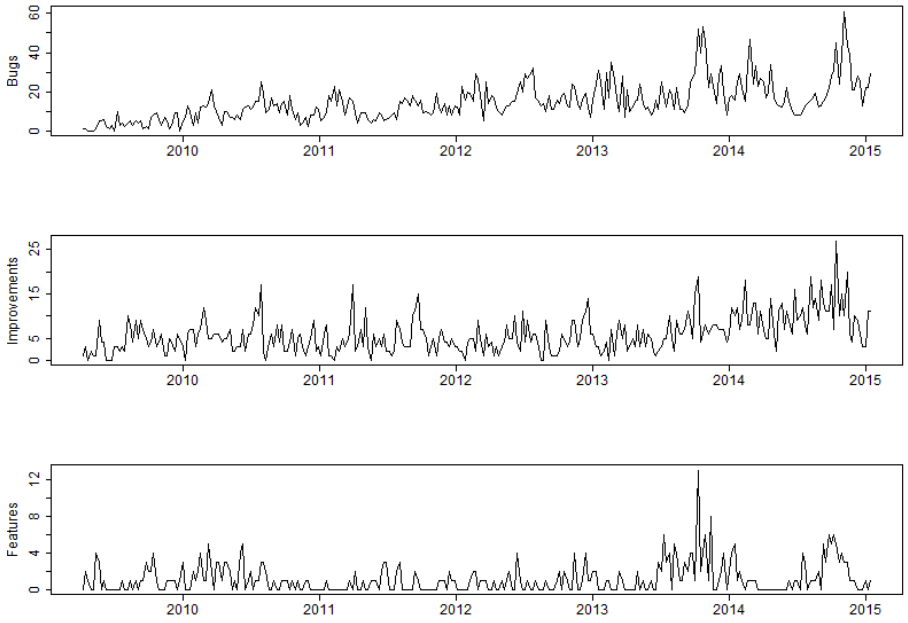


Figure 7.1: Time series data generated by sampling the *MongoDB* dataset with a 7-day sample period.

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 7.1. 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.

	Y_{bug}		Y_{imp}		Y_{new}	
Statistic	value	signif.	value	signif.	value	signif.
ADF (τ_2)	-5.0203	$< 1\%$	-7.4022	$< 1\%$	-7.8448	$< 1\%$
ADF (ϕ_1)	12.6505	$< 1\%$	27.4154	$< 1\%$	30.7709	$< 1\%$
KPSS	2.8521	$< 1\%$	2.0208	$< 1\%$	0.5269	$2.5 - 5\%$

Table 7.1: Results of running the ADF unit root test and KPSS stationarity test on Y_{bug}, Y_{imp}, and Y_{new}. The tests are ran using an intercept-only regression model.

After differencing we obtain the time series shown in figure 7.2, which will be referred to as $Y_{\bigtriangledown bug}$, $Y_{\bigtriangledown imp}$, and $Y_{\bigtriangledown new}$. Now the result of the unit root and stationarity test (listed in Table 7.2) 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.

	$Y_{\bigtriangledown bug}$		$Y_{\bigtriangledown imp}$		$Y_{\bigtriangledown new}$	
Statistic	value	signif.	value	signif.	value	signif.
ADF (τ_2)	-17.6529	$< 1\%$	-20.4382	$< 1\%$	-21.8989	$< 1\%$
ADF (ϕ_1)	155.8144	$< 1\%$	208.8647	$< 1\%$	239.7814	$< 1\%$
KPSS	0.0115	$> 10\%$	0.0127	$> 10\%$	0.0127	$> 10\%$

Table 7.2: Results of running the ADF unit root test and KPSS stationarity test on the differenced time series

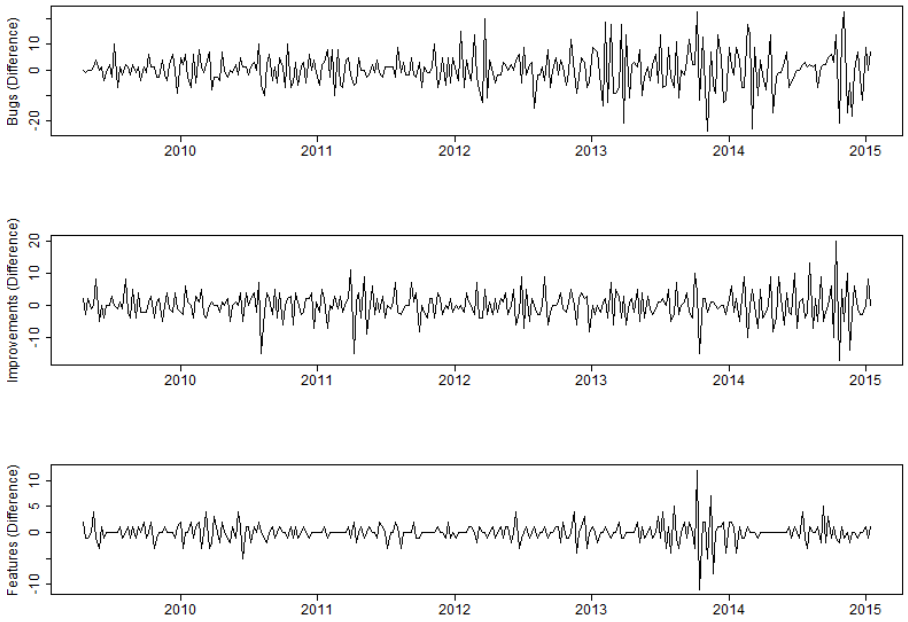


Figure 7.2: Differenced time series data.

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 W_{2-79}, W_{80-157}, and $W_{158-235}$.

Time Series Model

The modeling methodology was applied to the time series data, to produce a VARX model for each windowed period. The $Y_{\bigtriangledown new}$ and $Y_{\bigtriangledown imp}$ time series were both considered exogenous, so that their hypothetical future values alone could be used to make defect predictions.

Model Specification & Estimation

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

$$p_{max} = \left\lfloor \frac{78}{(3)(4)} \right\rfloor = \lfloor 6.5 \rfloor = 6 \tag{7.1}$$

So models of order 1 through $p_{max} = 6$ 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 7.3. 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 W_{2-79}, and order 5 for period $W_{158-235}$.

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

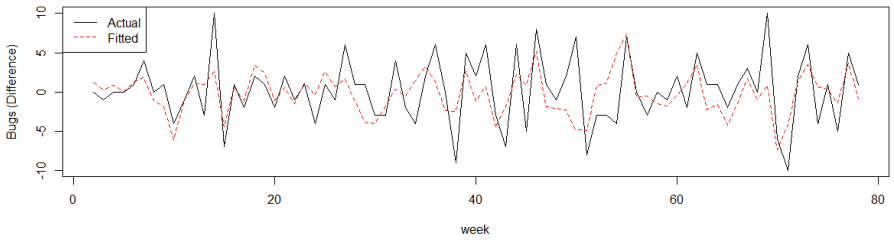
Table 7.3: Results of running stability and Ljung-Box test on each windowed period.

Model Selection

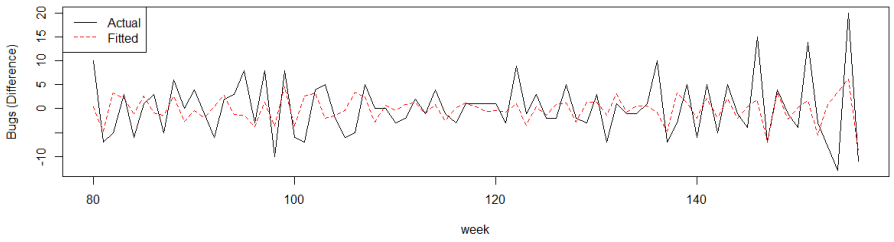
Models that were not rejected for instability or inadequacy were then compared and the best for each windowed period was selected by AIC selection criterion. The results of selection are the bolded values shown in Table 7.4. The fit for each of these models is demonstrated by plotting one-step predictions along with actual values, as shown for each model in figure 7.3. The fit for each appears to track well with many of the significant changes in the time series.

	AIC score		
Model order	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

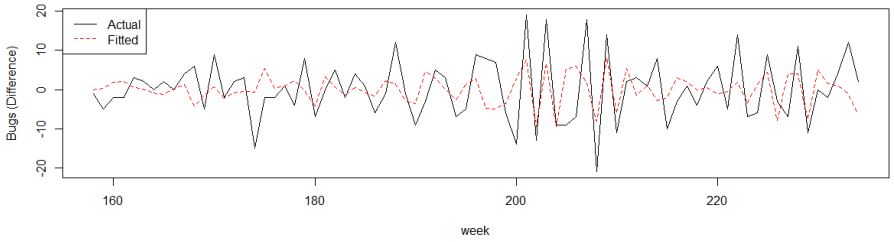
Table 7.4: Results of model selection, using AIC score to compare models of different order.



(a) Windowed period W_{2-79}



(b) Windowed period W_{80-157}



(c) Windowed period $W_{158-235}$

Figure 7.3: One-step predictions vs actual values, for each model selected by AIC score.

Forecasting

The model selected for each windowed period 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 7.5 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 bold outlined row in Table 7.5). On the other hand, the actual number of future defects in the next window, W_{80157}, was 17. This was inside the 90% confidence interval, which spans from 13.38 to 18.00.

Improvements	Features	90% low	75% low	Mean	75% high	90% high
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	9.1	11.69	12.8
4	0	6.4	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	6.2	7.31	9.89	12.48	13.59

Table 7.5: Forecasting at the end of the first time window, W_{2-79}. Future output values are predicted for a number of hypothetical input values.

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 distribu-

tion of errors between the mean forecasted bugs and the actual number of bugs is shown as a histogram in Figure 7.4. 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.

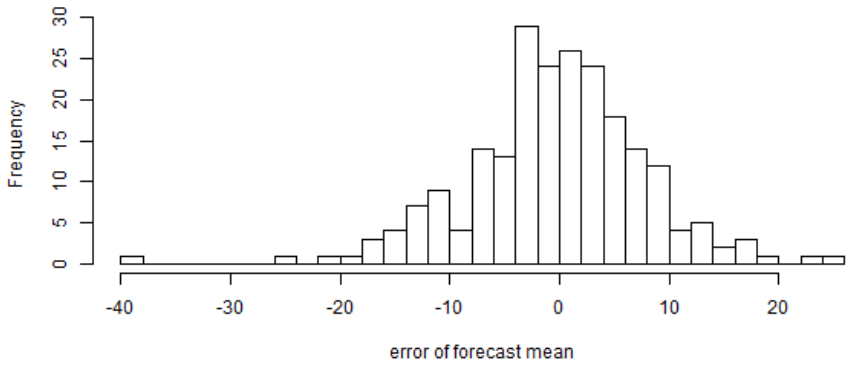


Figure 7.4: Histogram of forecast mean errors obtained using a 78-week sliding window.

Chapter 8

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 *Mozilla*, to better determine the applicability of the modeling approach. The estimated time to complete this additional work is one month.

Bibliography

[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.