Using Time Series Models for Defect Prediction in Software Release Planning

Thesis Proposal

James Tunnell

March 9, 2015

Introduction

In software release planning there are two primary concerns: functionality and quality. To improve functionality and maintain high quality are the common objectives. 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. Thus, a high quality of software can be ensured 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 schedule or quality slip. Since the time required for testing and bug-fixing will likely be a function of the number of defects introduced during development, it would be desirable to have a technique for predicting how many bugs can be expected as development proceeds.

Many software defect prediction techniques rely on code analysis. Other techniques rely on statistical modeling using empirical time series data. Both approaches are discussed in the Related Work section. In this paper, a time series method is used (see the Time Series Modeling section).

One potential application of a defect prediction model is for comparing different release plans, to see how much bug fallout would likely result. This would help planners compare release plans to ensure that total development time does not exceed the time budget. The comparison of different plans is integral to release planning optimization, which is the focus of The Next Release Problem, a key problem in Search-Based Software Engineering (SBSE). This application of prediction models is discussed in the Motivation section.

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 statistical models discussed in the Related Work section are limited in this respect, as they depend only on the past defects. For this reason, it is proposed that for this application, a model be used that depends both on past defects and on planned features and improvements. Specifically, use of a multivariate time series model that includes exogenous inputs. Such a model is presented in the Time Series Modeling section.

The model is then applied to data from the MongoDB software project, to see which model structure is selected and how well it predict defects. The results of this are shown in the Results section. Additional work is then proposed in the Proposed Work section.

Motivation

When software releases are planned in the usual way, then it is reasonable to construct a statistical predictive model that depends only on previous defects occurrences. After all, planned features and improvements will probably be selected in the same manner for the next release as in previous releases. So why not assume that defect occurrences in the next release will occur in like manner as in previous releases?

This assumption makes sense under normal planning conditions. But what if release planners wished to put together multiple release plans, and predict defect occurrences for each? Surely the predicted number of defects should not be the same for all release plans. Yet this would be the case if the predictive model depended only on previous defect occurrences (Figure 1 illustrates this limitation).

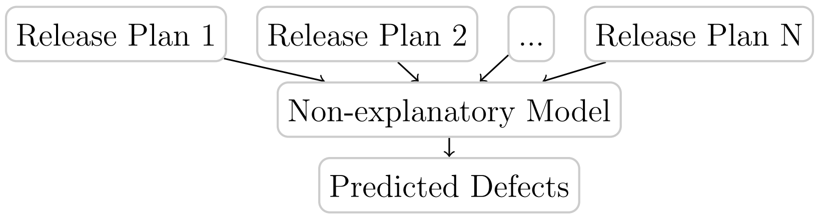


Figure 1: Using a non-explanatory model would result in the same defect prediction, regardless of the release plan.

Instead, to support these "what-if" scenarios, a predictive model should depend also on the basic elements of the release plan: the planned features and improvements. Such a model would assume some explanatory relationship, so that planned features and improvements somehow affect the outcome (see figure 2 for an illustration). If such a model were used, this would give release planners a path towards evaluating the additional development time needed to address bug fallout, for a given release plan. By ensuring sufficient time to fix bugs, the model can be used to ensure sufficient software quality is maintained, thus giving the release planner freedom to otherwise maximize the expected revenue produced by the software by including appropriate features and improvements.

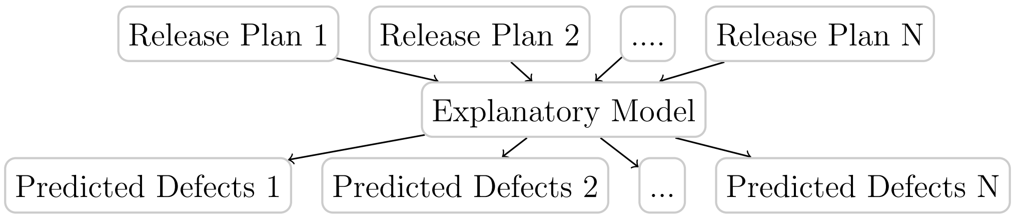


Figure 2: Using an explanatory model allows for the possibility of different defect predictions for each release plan.

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 a 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 et al[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 is designed to aid software project planners, who have multiple customers to satisfy. The project planner 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

$$\hat{R}_i = R_i \cup ancestors(R_i) \tag{1}$$

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

$$cost(R') = \sum \{cost(r) | r \in \hat{R}_i\} \tag{2}$$

Once customer i is satisfied, their weight w_i contributes to the total revenue from the project, as in

$$\sum_{i \in S} w_i \tag{3}$$

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

$$\begin{aligned} &maximize \quad \sum_{i \in S} w_i \\ &subject\ to \quad cost(\bigcup_{i \in S} \hat{R}_i) \leq B \end{aligned} \tag{4}$$

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.

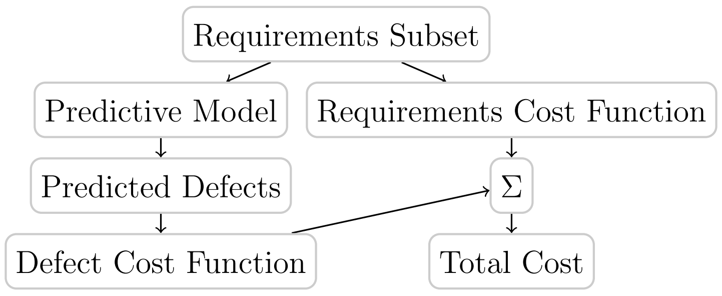


Figure 3: Defect prediction model used in determining overall cost of some requirements subset.

Since predictive models can not 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.

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 et al.[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 et al.[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 et al [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".

Time Series Modeling

In this section, time series data and models are discussed.

Time Series

A time series is a collection of observations occurring in order. The process underlying a time series is assumed to be stochastic, so a model must be probabilistic. Critically, the sequence of observations can not be re-arranged, because each observation is typically dependent somehow on previous observations. It is this dependence that complicates the modeling of time series data, because otherwise observations would be independent and values would simply follow some probability distribution.

Autocorrelation, ACF, and PCF

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

ARMA and ARIMA (Univariate) Models

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

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

Endogeneity and Exogeneity

Exogenous variables are not is not considered to be under the ”control” of the model, and instead should be considered an input. As such, a model should only try to account for the variable’s behavior using its previous values, and not using previous values of other variables. A model should still establish interdependence between all variables, endogenous or exogenous. For exogenous variables, past values of all variables can be used to predict future values.

VAR and VARX (Multivariate) Models

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

However, these vector models so far have no way to account for non-stationary time series. So, as a preliminary step to using them, the time series data should either be stationary already, or differenced to become stationary. Trends and tests for stationarity will be discussed next.

Trends

Trending time series are challenging to analyze, because the summary statistics of mean, variance, and autocovariance will vary over time, and are therefore not interpretable.[5]. Two trend types are discussed: deterministic and stochastic. A deterministic trend will be moving upward or downward, so the time series mean is non-constant, but it will be according to a deterministic function. In this case, time series movements will follow generally the deterministic function, with non-permanent fluctuations above or below. Such a time series is said to be stationary around a deterministic trend. In contrast, a stochastic trend shows permanent effects whenever random shocks occur, not necessarily fluctuating only close to the area of deterministic function. Differencing is applied to remove a stochastic trend. In the following section, tests are discussed for determine whether a deterministic or stochastic trend is present.

Stationarity Tests

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.

Unit Root 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 found to be significant, the null hypothesis can not 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.

Stationarity Testing

On the other hand, a stationarity test uses the null hypothesis that a time series is stationary around a deterministic trend, with the alternative that Then, 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 KPSS test is often applied for testing stationarity.

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 data used so far comes from the MongoDB Core Server project, which has been ongoing since May of 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 data using a Python[2] script.

[1]JIRA is an issue tracking and project management system made by Atlassian, who provides a free JIRA subscription for qualified open source projects.
[2]Python is a dynamic, general purpose programming language.

The following fields are kept from each issue: type, priority, creation date, resolution date. Once extracted, the data is changed to text table format, suitable for reading in R[3].

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. Fixed issues are predominant, so there is little risk in this decision. 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

Once read into an R script, the data is operated on to prepare it for time series modeling. The data will be sampled, made stationary, and windowed. These steps are discussed next.

Sampling

First, the data is sampled by dividing time into sample periods. In each period, the data is sampled to measure: the number of improvements resolved, features resolved,

[3]R is a popular software environment for statistical computing.

and bugs created. As an example, this sampling process is illustrated in figure 4, with results shown in Table 1.

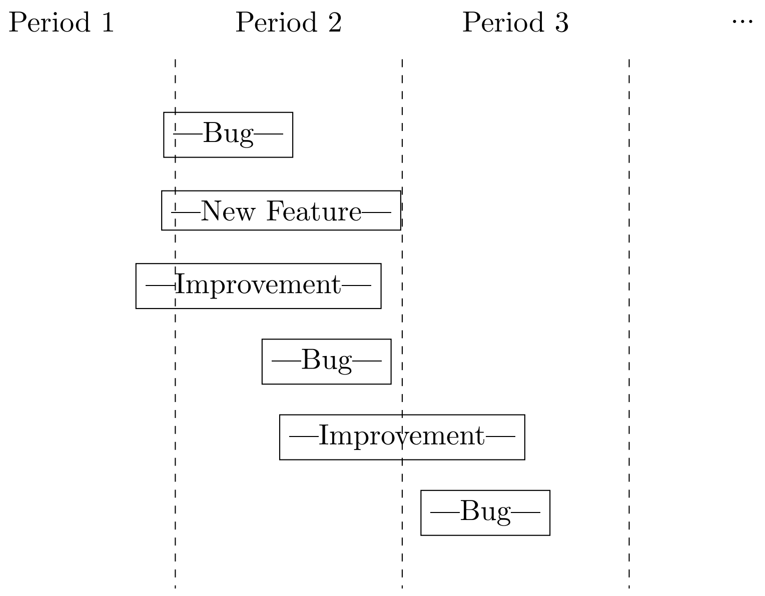


Figure 4: Sampling issue data by dividing time into equally-spaced periods.

Table 1: Results of sampling example issues shown in figure 4.

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

Establishing Stationarity

The importance of stationarity has already been discussed in the Time Series Modeling section.

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 can not be ruled out, then KPSS test should be applied to check that trend stationarity is also rejected. If data is found to be have a stochastic trend, it should be differenced and then retested to confirm (trend) stationarity.

The the *urca*[4] library provides `ur.df` and `ur.kpss` functions for performing these test. 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 SW development process underlying a given project may 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. The time window should balance between more observations, to capture consistent long-term behaviors, and less observations, to limit exposure inconsistent short-term behaviors.

Taking this approach means that the entire modeling methodology will be executed for each time-windowed part of the data.

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].

[4]The urca library provides tests for time series data, and is freely available as a package for the *R* computing environment.

Model Specification & Estimation

Specification of 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 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 their 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

$$K = \frac{mn}{m^2p} = \frac{n}{mp} \tag{5}$$

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

$$K_{min} \leq K = \frac{n}{mp} \tag{6}$$

In terms of p this becomes

$$p \leq \frac{n}{mK_{min}} \tag{7}$$

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

$$p_{max} = \left\lfloor \frac{n}{mK_{min}} \right\rfloor \tag{8}$$

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.

To estimating the parameters of a VARX model, the *dse*[5] library provides the `estVARXar` function.

Diagnostics Checking

To verify that a model can be accepted, diagnostic checking is performed. Included in this step is testing for stability and for model inadequacy.

Stability Tests

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. The *dse* library provides the `stability` function for performing this test.

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. These lack-of-fit tests are a kind of portmanteau test.

One of these tests, the Ljung-Box test, forms a statistic from the autocorrelation of the residuals (up to some lag). In this test, the null hypothesis is that residuals are independent, so their autocorrelation is not high enough to be distinguished from a white noise series. To support this hypothesis, the test p-value should be above some level of significance, say 5%. The *stats*[6] library provides the `Box.test` function for performing the Ljung-Box test.

[5]The dse library provides tools for time series models, and is freely available as a package for the *R* computing environment.

[6]The stats library provides core statistics functions, and is freely available as a package for the *R* computing environment.

Model Selection

Model selection criteria are used to compare models by their fit, to minimize residual error, and penalizing the model to some degree by the number of parameters. In the case of Akaike Information Criterion (AIC), for ARMA models in general:

$$AIC_{p,q} = ln|\tilde{\mathbf{\Sigma}_r}| + 2r/n, \tag{9}$$

where n is the number of samples, r is the number of parameters, and $\tilde{\mathbf{\Sigma}_r}$ is the residual covariance matrix estimate. AICc and BIC are other well-known selection criteria. But, according to Bisgaard and Kulahci, "The 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 will be used as the selection criterion.

The *dse* library provides the `bestTSestModel` function for performing model selection.

Results

Collecting data

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

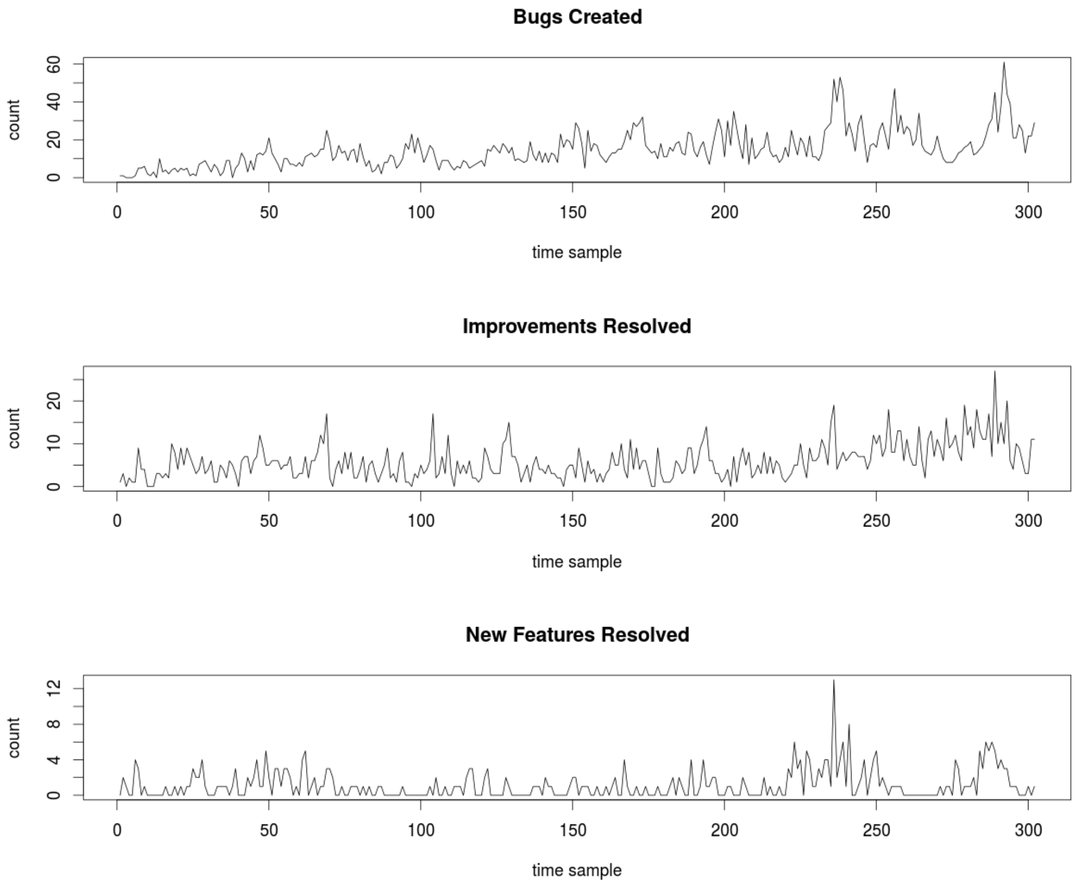


Figure 5: Time series data generated by sampling the MongoDB dataset with a 7-day sample period.

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 tables 2 and 3, respectively. Remember that for these tests, we need low significance for the unit root test and high significance for the stationarity test. This will allow us to reject the hyopthesis that a time series has a unit root (is not stationary) and fail to reject the hypothesis that the time series is stationary.

The unit root tests showed less than 1% significance for all time series. However, the stationarity test also showed low significance. Since there is disagreement in the test results, the time series will all be differenced and the tests re-run.

	Y_{bug}		Y_{imp}		Y_{new}	
Statistic	value	signif.	value	signif.	value	signif.
τ_2	-5.0203	$< 1\%$	-7.4022	$< 1\%$	-7.8448	$< 1\%$
ϕ_1	12.6505	$< 1\%$	27.4154	$< 1\%$	30.7709	$< 1\%$

Table 2: Results of running the augmented Dickey-Fuller unit root test on Y_{bug}, Y_{imp}, and Y_{new}. The test is ran using an intercept-only regression model.

Time series	statistic	signif.
Y_{bug}	2.8521	$< 1\%$
Y_{imp}	2.0208	$< 1\%$
Y_{new}	0.5269	$2.5 - 5\%$

Table 3: Results of running the KPSS stationarity test on Y_{bug}, Y_{imp}, and Y_{new}. The test is ran assuming a constant-only deterministic component.

After differencing we obtain the time series shown in figure 6, 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 tables 4 and 5) both agree. That is, we can reject the hypothesis that a unit root is present at the 1% significance level and we have significance to accept stationarity with greater than 10% significance. Hence, the differenced time series will be used to move forward with modeling.

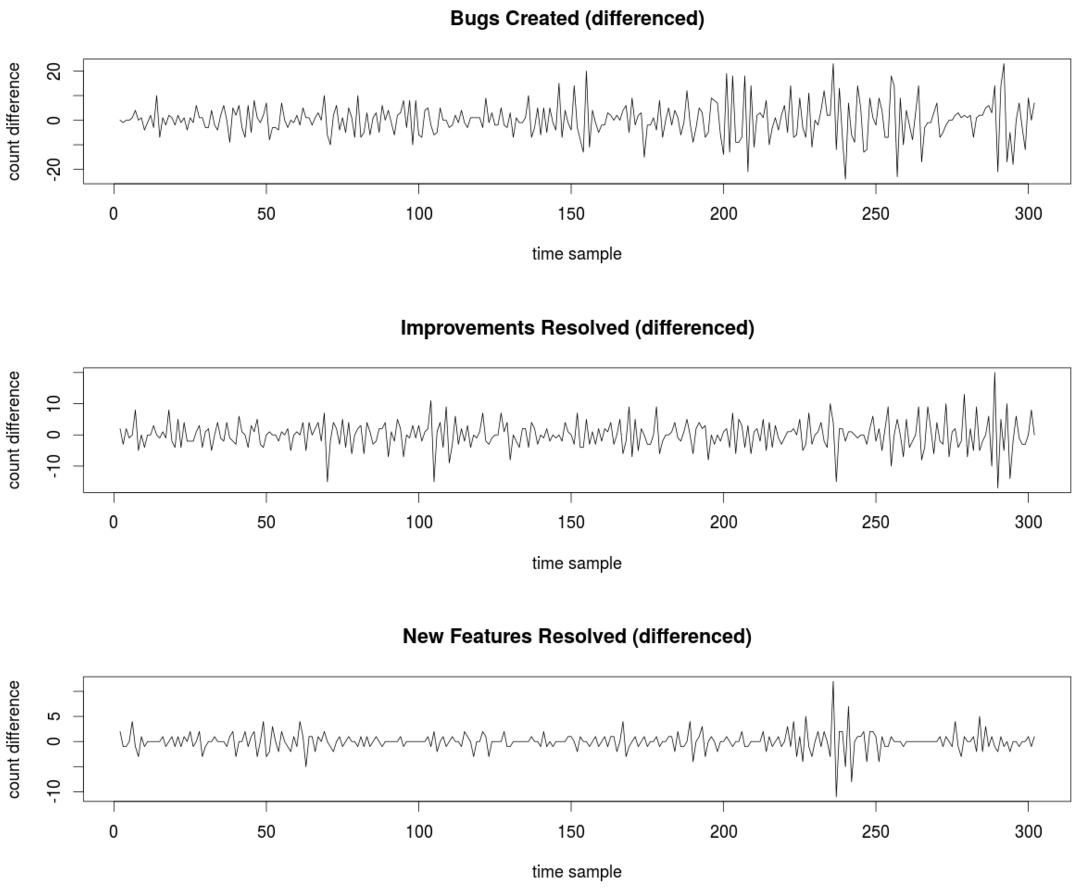


Figure 6: Differenced time series data.

	$Y_{\bigtriangledown bug}$		$Y_{\bigtriangledown imp}$		$Y_{\bigtriangledown new}$	
Statistic	value	signif.	value	signif.	value	signif.
τ_2	-17.6529	$< 1\%$	-20.4382	$< 1\%$	-21.8989	$< 1\%$
ϕ_1	155.8144	$< 1\%$	208.8647	$< 1\%$	239.7814	$< 1\%$

Table 4: Results of running the augmented Dickey-Fuller unit root test on the differenced time series

Time series	statistic	signif.
$Y_{\bigtriangledown bug}$	0.0115	$> 10\%$
$Y_{\bigtriangledown imp}$	0.0127	$> 10\%$
$Y_{\bigtriangledown new}$	0.0127	$> 10\%$

Table 5: Results of running the KPSS stationarity test on the differenced time series

Time Windowing

An 78-week time window (approximately 18 months) was established to restrict model scope. Three of these windowed periods, non-overlapping, were kept for modeling, and will be denoted W_{1-78}, W_{79-156}, and $W_{157-234}$.

Time Series Model

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

Model Specification & Estimation

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

$$p_{max} = \left\lfloor \frac{78}{(3)(4)} \right\rfloor = \lfloor 6.5 \rfloor = 6 \tag{10}$$

Models of order $1, 2, ..., p_{max}$ 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 6. All model orders were stable for all windowed periods. Several model orders were forund to be inadequate by the Ljung-Box test: orders 1-2 for period W_{1-78}, and order 5 for period $W_{157-234}$.

	W_{1-78}		W_{79-156}		$W_{157-234}$	
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

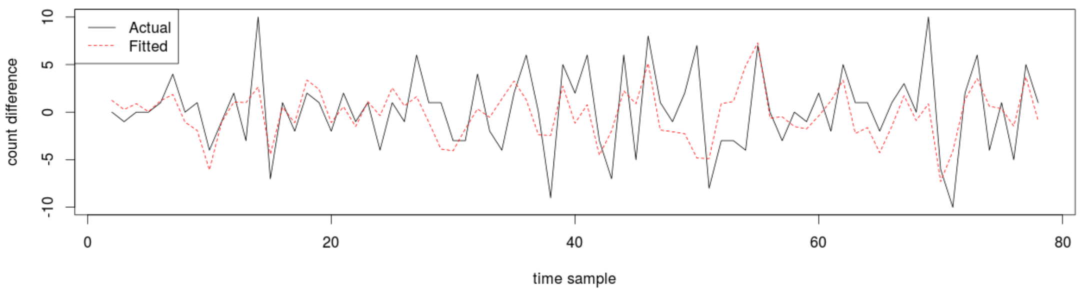
Table 6: Results of running stability and Ljung-Box test on each windowed period.

Model Selection

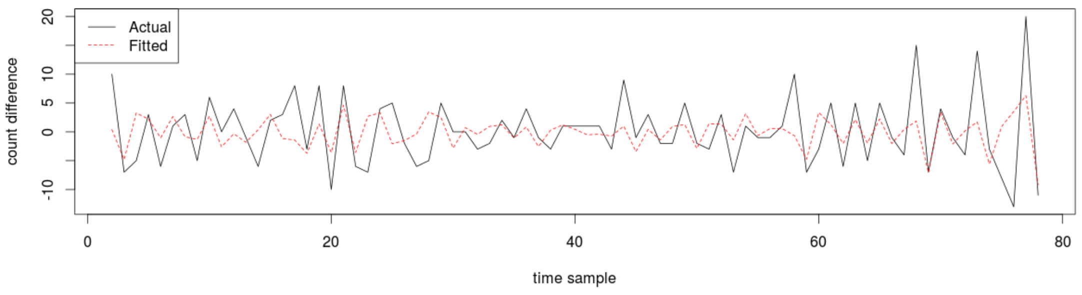
Models that were not rejected for instability or inadequacy were then compared and the best for each windowed period was selected by AIC selection criterion. The results of selection are shown in table 7, with orders 4, 1, and 1 being chosen for periods W_{1-78}, W_{79-156}, and $W_{157-234}$, respectively. The fit for each of these models is demonstrated by plotting one-step predictions along with actual values, shown for each model in figure 7.

Model Hypothetical scenarios

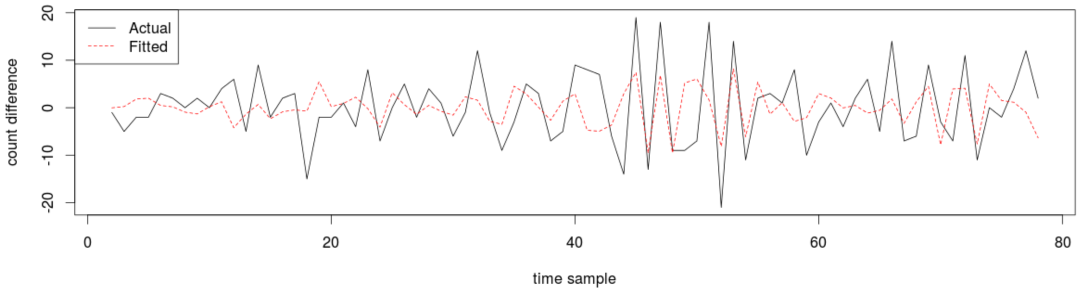
TODO



(a) Windowed period W_{1-78}



(b) Windowed period W_{79-156}



(c) Windowed period $W_{157-234}$

Figure 7: One-step predictions vs actual values, for each model selected by AIC score.

	AIC score		
Model order	W_{1-78}	W_{79-156}	$W_{157-234}$
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: Results of model selection, using AIC score to compare models of different order.

Proposed Work

After applying the data and modeling methodologies to the MongoDB dataset, a model is in-hand for each windowed time period. However, to adequately answer the question of whether such models can be used in release planning, further still remains. This proposed work is as follows:

1. Apply the model to hypothetical "release plans", that is, predict future values of Y_{bug} using hypothetical future values of Y_{imp} and Y_{new}. Keeping in mind the prediction confidence intervals, this would indicate how such a model might be in release-planning.

2. Follow the same methodologies with two other SW projects.

The time estimated to complete the proposed work is summarized in the timeline shown in table 8.

Task	Date
Form committee	Mar 9-13
Present proposal	Mar 18
Respond to committee feedback	Mar 18 - Apr 8
Apply models for release planning	Apr 1-15
Repeat on two other SW projects	Apr 15 - May 6

Table 8: Timeline for proposed work