Using Time Series Models for Defect

Prediction in Software Release Planning

James Tunnell, John Anvik

Computer Science Department

Central Washington University

Ellensburg, WA 98926, USA

[tunnellj, janvik]@cwu.edu

*Abstract*—To maintain a high-quality software release, sufficient time should be allowed for testing and fixing defects. Otherwise, there is a risk of schedule or quality slip. To this end, a time series model is developed for predicting the number of defects detected during development. The model depends on previous values of: defects created, new features resolved, and improvements resolved. This model structure supports defect prediction using hypothetical future values of new features resolved and improvements resolved, meaning hypothetical release plans could be compared according to their predicted impact on testing and defect-fixing time.

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

#### 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. In this paper, a time series model is used.

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).

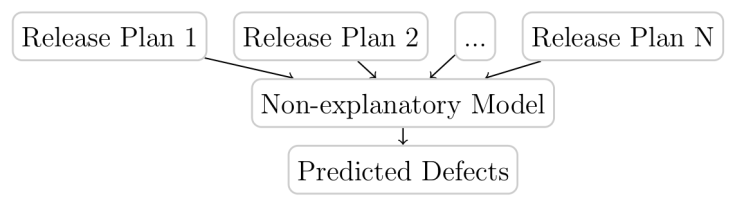
To make the defect prediction model useful for comparing release plans, the model must be dependent in some way on the basic elements of the release plan: planned new features and improvements. For this reason, it is proposed that for this application, a model be used that depends both on past defects and on planned features and improvements. Specifically, use of a multivariate time series model that includes exogenous inputs.

The model is then applied to data from the *MongoDB*[[1]](#footnote-1) software project, to see which model order is selected and how well it predict defects.

#### Motivation

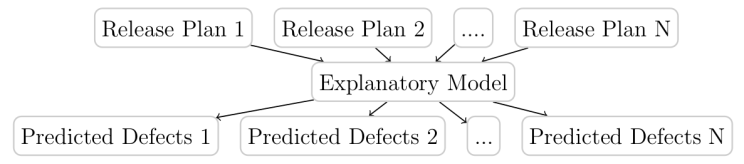
When software releases are planned in a way that is consistent with previous releases (in the same project), then it is reasonable to construct a statistical predictive model that depends only on previous defects occurrences. After all, planned features and improvements will probably be selected in a similar manner for the next release as they were 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, where planners are likely relying on their experience and on established conventions to arrive at a satisfactory set of requirements for an upcoming release. But what if release planners instead opt to employ a heuristic or optimization to make their decision? This would require the comparison of multiple hypothetical release plans. In this case, we would expect the predicted number of defects to not necessarily be the same for all release plans. Yet this would be the case if the predictive model depended only on previous defect occurrences (Fig. 1 illustrates this limitation).



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 Fig. 2).



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

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.

## 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. Briefly, the problem is posed as follows:

* A software project has some set of requirements to consider for implementation in the next release
* A customer will provide revenue when a particular subset is included in the release
* Requirements may have dependencies on other requirements as well
* Each requirements has an associated cost
* Total cost is the sum of costs for all requirements included for the next release, including dependencies
* Total cost must be below some budgeted amount

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

## The Gap Between Abstraction and Reality

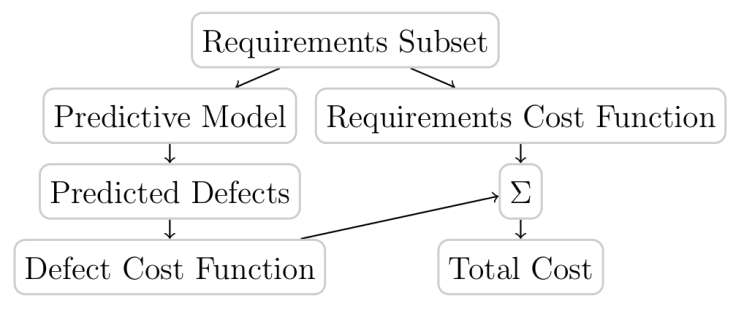
As was discussed in the previous section, a planner would need several things to be able to implement a NRP-like optimization:

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, as depicted in Fig. 3.



1. 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.

#### Related Work

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

## Code Analysis Approaches

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

## Statistical Approaches

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

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

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

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

#### 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 cannot 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. 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 not try to account for an exogenous variable’s behavior, but instead use past values to predict the behavior of other, endogenous (non-exogenous) variables.

## VAR and VARX (Multivariate) Models

By extending the ARMA model to the multivariate case, 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 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.

## Stationarity 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.

#### 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[[2]](#footnote-2) 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[[3]](#footnote-3) script.

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[[4]](#footnote-4).

## 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

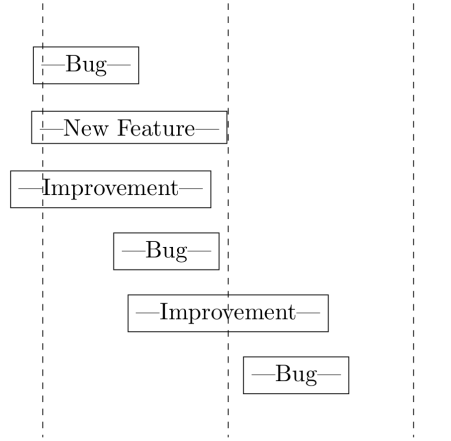
Issues that are sub-tasks are first converted to be the same type as the parent issue. Those sub-tasks whose parent issue is not in the dataset are considered orphans and discarded. There were 20 (0.28%) orphaned sub-tasks encountered in the dataset.

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



1. Sampling issue data by with equally-spaced periods.
2. Results of sampling example the issues shown in Fig. 4.

|  |  |  |  |
| --- | --- | --- | --- |
| 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.

The *urca[[5]](#footnote-5)* 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 software 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 fewer observations to limit exposure to 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, p. 581].

## Model Specification & Estimation

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

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

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

In terms of *p* this becomes

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

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

To estimate the parameters of a VARX model, the *dse[[6]](#footnote-6)* 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 model inadequacy tests are formed around a study of the residuals. These lack-of-fit tests are a kind of portmanteau test.

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

## 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:

,

where is the number of samples, is the number of parameters, and is the residual covariance matrix estimate. AICc (AIC with correction) and BIC (Bayesian Information Criterion) 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

## Data Collection

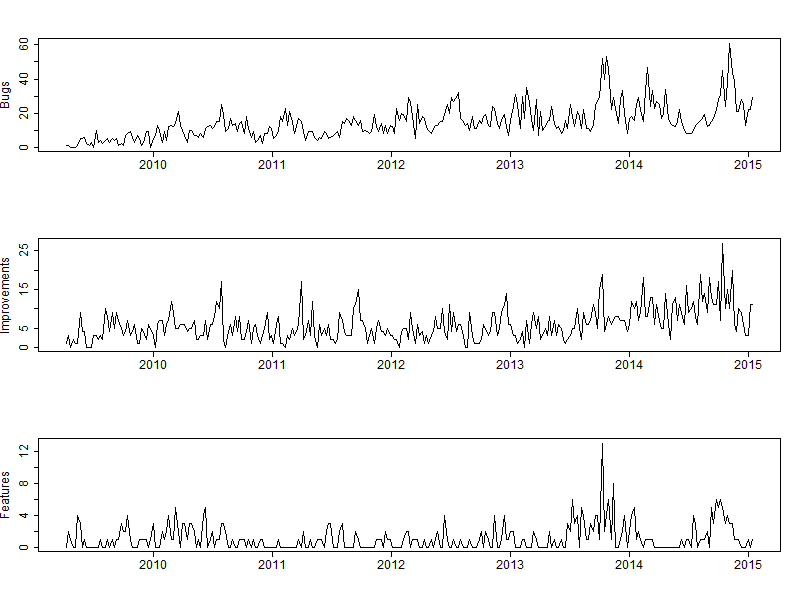
The *MongoDB* dataset was collected according to the methodology in the Data Methodology section, and the data set was sampled with a 7-day sample period to create the following time series: bugs created, improvements resolved, and new features resolved. These time series will be denoted , , and , respectively,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

## Stationarity Testing

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

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

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



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

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

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

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

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



1. Differenced time series data.

## Time Windowing

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

## Time Series Model

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

## Model Specification & Estimation

By selecting , a maximum model order is obtained by

Models of order were estimated for later diagnostic checking.

## Model Diagnostic Checking

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

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

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

## Model Selection

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

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

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

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

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

#### Conclusions and Future Work

##### Acknowledgment

##### References

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

1. *MongoDB* is a document-oriented, NoSQL database, and is available under the [GNU Affero GPL](https://gnu.org/licenses/agpl.html) license. [↑](#footnote-ref-1)
2. JIRA is an issue tracking and project management system made by Atlassian, who provide free JIRA subscriptions for qualified open source projects. [↑](#footnote-ref-2)
3. Python is a dynamic, general purpose programming language. [↑](#footnote-ref-3)
4. R is a popular software environment for statistical computing. [↑](#footnote-ref-4)
5. The [*urca*](http://cran.r-project.org/web/packages/urca) library for R provides tests for time series data. [↑](#footnote-ref-5)
6. The *dse* library for R provides tools for time series models [↑](#footnote-ref-6)
7. The *stats* library for R provides core statistics functions. [↑](#footnote-ref-7)