A Defect Prediction Model for Software Release Planning

Thesis Proposal

James Tunnell

February 28, 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).

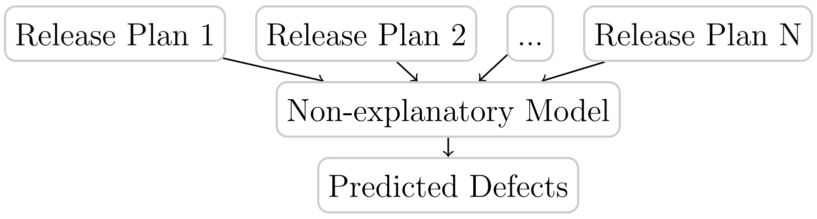

Figure 1: Using a non-explanatory model would result in the same defect prediction, regardless of the release plan.

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.

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

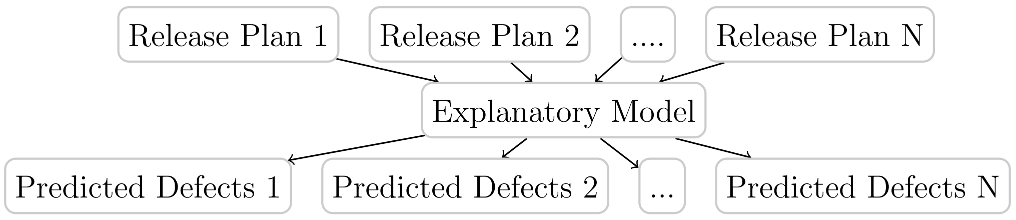

Figure 2: Using an explanatory model allows for the possibility of different defect predictions for each release plan.

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[9, 14, 16].

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.

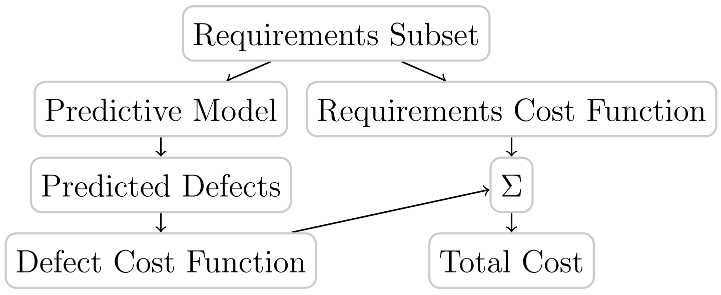

Figure 3: Defect prediction model used in determining overall cost of some requirements subset.

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 [5] likewise predicted defect count based on LOC. Rather than code itself, Henry and Kafura [8] define metrics that are based on information taken from design documents, to be used in defect prediction. Nagappan and Ball [12] use relative code churn (lines modified) as a metric for predicting the density of defects. Giger, Pinzger, and Gall [6] 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.[10], 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.[7] 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 [13], 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 modeling residuals (difference between actual and fitted values) to check for statistical significance.

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"[3]. 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 PCF 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

An exogenous variable 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.[4]. 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[11]. Alternatively, a weak stationarity (of second order) can be established, and from this strict stationarity can be established by then assuming normality[3].

For a multivariate time series, stationarity holds if all the component univariate time series are stationary.[15], 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[4]. 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 is accepted, and it is established that the time series has a stochastic trend and is therefore non-stationary. The augmented Dickey Fuller regression 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.

Modeling Methodology

A typical methodology for "...the building of an appropriate vector ARMA model..." is outlined by Box et al[3, p. 478] as follows:

1. Model specification/identification
2. Model parameter estimation
3. Model diagnostics checking

In addition to these steps, and as mentioned previously, data stationarity should be established before using the VARX model. This step is discussed next, and then the usual step in modeling methodology are discussed.

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 test. If we can indeed rule out a stochastic trend, we can 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. At that point, time series data should be differenced, and then retested to confirm (trend) stationarity.

Model Specification/Identification

As discussed by Box et al [3, pg. 581], one of the approaches for model specification is the use of model selection criteria. A maximum likelihood (ML) procedure used for this. In the case of Akaike Information Criterion (AIC), for ARMA models in general:

$$AIC_{p,q} = ln|\tilde{\mathbf{\Sigma}_r}| + 2r/n, \tag{5}$$

where n is the number of samples, r is the number of parameters, and $\tilde{\mathbf{\Sigma}_r}$ is the residual covariance matrix estimate.

Model Parameter Estimation

Model estimation is performed using a likelihood method, such as ML. Box et al also discuss the alternative method which uses the innovations form of likelihood, in which a linear predictor is formed from one-step predictions.

Diagnostics Checking

TODO

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.

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" 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.

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

Data Preparation

Once read into an R script, the data can easily be operated on to prepare it for time series modeling. First, a time window is established and divided into sampling 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 Figure 4, with results shown in Table 1.

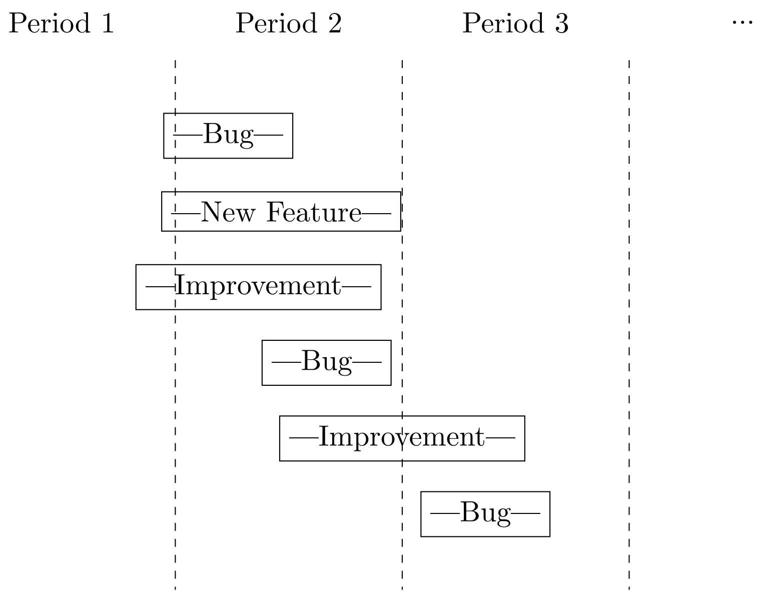

Figure 4: Sampling issue data by dividing time into equally-spaced periods.

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

Results

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.

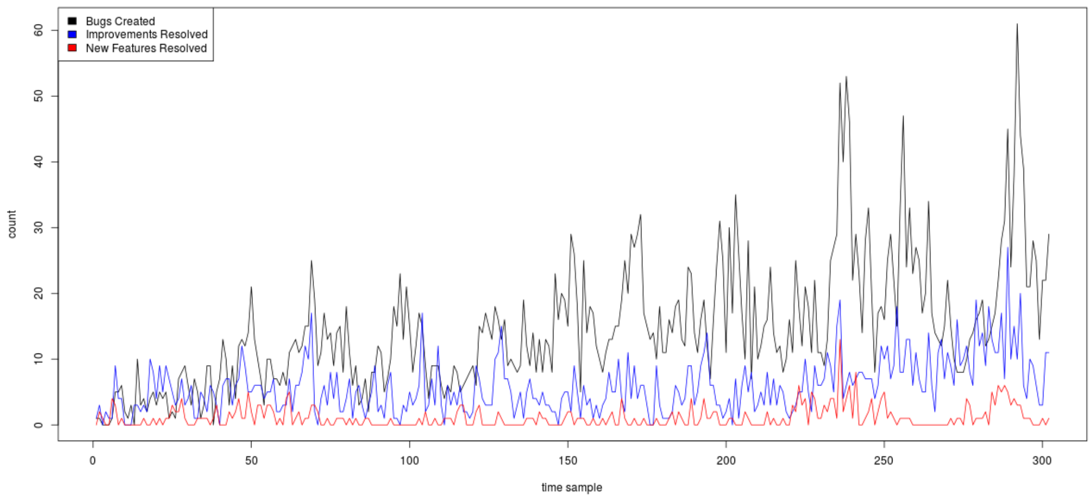

Figure 5: Time series data generated by sampling the MongoDB dataset with a 7-day sample period.

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.

For model specification and parameter estimation, the `bft` function of the *dse*[4] library was used. This function performs specification using model selection criteria for increasing AR order, and model parameter estimation using the innovations likelihood method

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

mentioned in the Modeling Methodology section. The results of applying this function on the MongoDB time series data are shown in Figure 6.

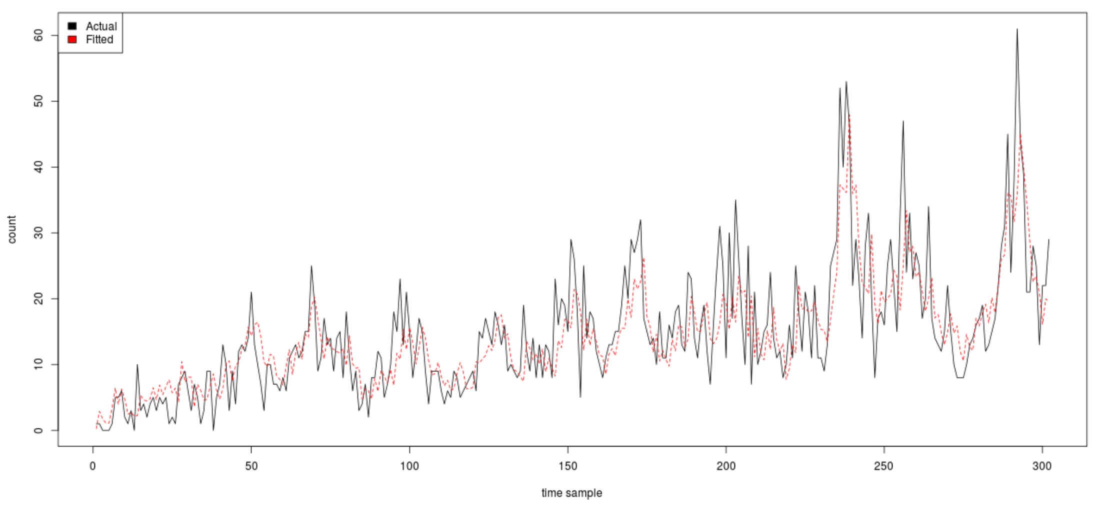

Figure 6: Results of model fitting shown by one-step predictions.

Lastly, the effects of hypothetical future values of Y_{new} and Y_{imp} are considered. Such hypothetical future values would be considered in the context of a software release plan. Different plans can be compared by their predicted values of Y_{bug}. The result of one-step Y_{bug} predictions for hypothetical Y_{new} and Y_{imp} values are shown in Figures 7 and 8. We see that using the VARX model, the one-step prediction case results in a linear relationship between hypothetical exogenous values and the predicted endogenous value.

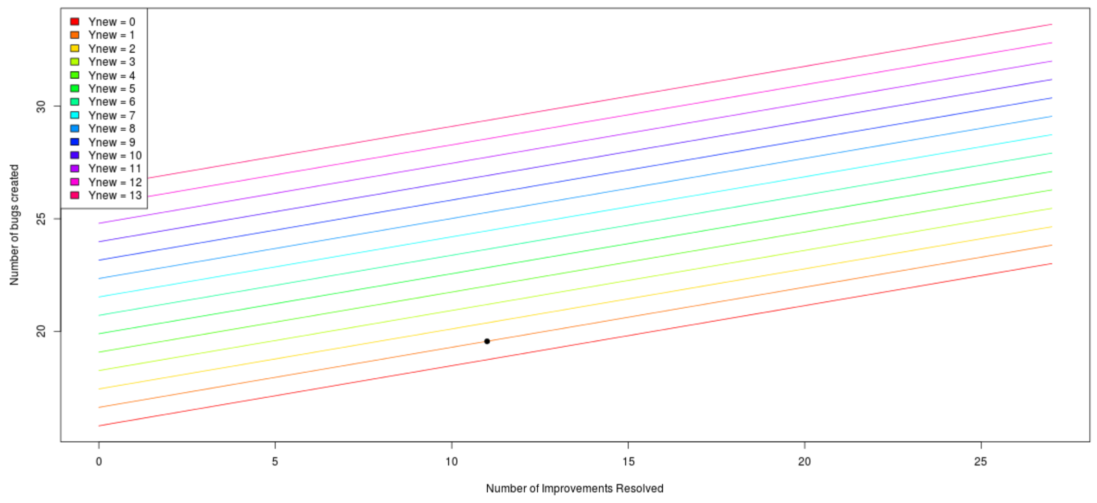

Figure 7: The effect of hypothetical future values for Y_{new} and Y_{imp}, shown as a series of lines in 2D.

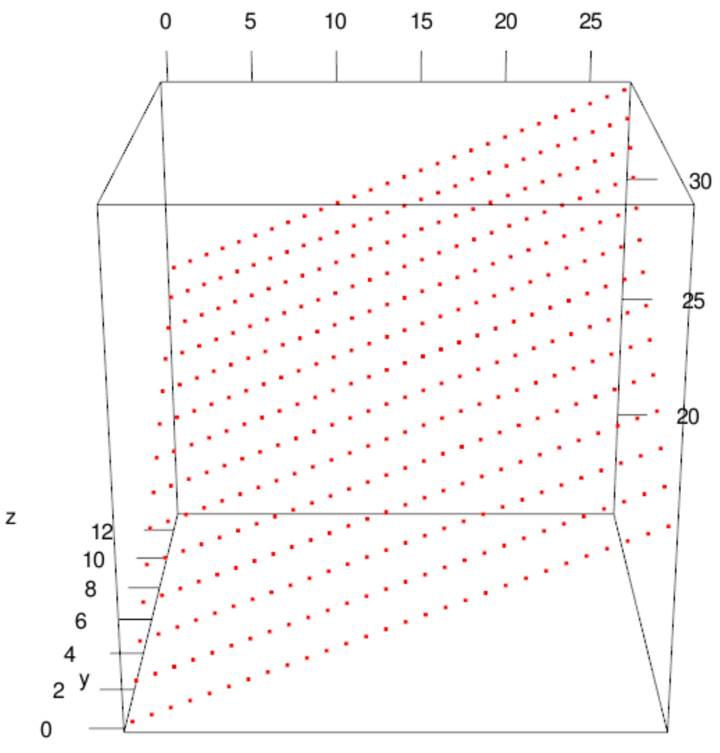

Figure 8: The effect of hypothetical future values for Y_{new} and Y_{imp}, shown as a plane in 3D.

Proposed Work

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] G. E. P. Box, G. M. Jenkins, and G. C. Reinsel. *Time Series Analysis.* John Wiley, 2008.

[4] P. H. Franses. *Time series models for business and economic forecasting.* Cambridge university press, 1998.

[5] J. E. Gaffney. Estimating the number of faults in code. *Software Engineering, IEEE Transactions on*, SE-10(4):459–464, July 1984.

[6] 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.

[7] 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.

[8] S. Henry and D. Kafura. The evaluation of software systems' structure using quantitative software metrics. *Software: Practice and Experience*, 14(6):561–573, 1984.

[9] 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.

[10] 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.

[11] T. K. Moon and W. C. Stirling. *Mathematical methods and algorithms for signal processing*, volume 1. Prentice hall New York, 2000.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] 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.