CHAPTER I

# Introduction

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

A critical step in this planning process is to factor in a suitable amount of time for testing and bug-fixing. Otherwise, there is a risk of slip in the development schedule and/or software quality. As the time and effort required for testing and bug-fixing will likely be a function of the number of defects introduced during development, it is desirable to be able to predict how many bugs can be expected as development proceeds.

A potential application for defect prediction is to compare different release plans according to their estimated bug fallout and subsequent impact on testing and bug-fixing times. This would assist release planners in ensuring that the total development time does not exceed the project’s time budget for a release. The comparison of different release plans is integral to release plan optimization, which is the focus of The Next Release Problem (discussed in detail in the Motivation chapter).

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

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

The paper is organized as follows. First, related work is presented in the Literature Review chapter. Then, further motivation for the use of a time series model for predicting defects is presented in the Motivation section. Next, an overview of time series modeling concepts is provided in the Background section. The methods used for data collection and preparation, and for time series modeling are detailed in the Methods chapter, respectively. The results of applying these methods are then given in the Results chapter. Last, the paper concludes and poses future work in the Conclusions & Future Work chapter.

CHAPTER II

# Literature Review

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

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

## Statistical Approaches to Defect Prediction

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

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

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

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

CHAPTER III

# Motivation

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

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

An alternative approach is to develop a model that will take into account the differences in composition of features and improvements between the release plans. In this case, one would expect that the predicted number of defects would vary across the release plans and that prediction uncertainty can be quantified by confidence intervals. Such a model would assume some explanatory relationship, like that shown in Figure 1.



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

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

## Application to the Next Release Problem

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

### Defining the NRP

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

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

A software project has a set of all possible requirements (new features and enhancements) that might be included in the next software release. A customer is satisfied by completing a subset . The importance of a customer is given by the weight, .

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

For example, if , and is a prerequisite for , then .

A requirement has a cost , associated with its implementation, not considering the cost of any prerequisite requirements. Then, the cost for some subset will be

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

So, the NRP is posed as follows: for a group of customers, select the subset that maximizes total revenue, while keeping the total cost within some budget constraint . This is given by

### The Gap Between Abstraction and Reality

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

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

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

### Bridging the Gap

We use the explanatory model to address the need to consider defect cost. Such a model, given some subset of proposed requirements, can be used to predict defects and to find additional cost which should be considered. This use of the predictive model is illustrated in Figure 2.



Figure Defect prediction model being used to determine the overall cost of some requirements subset.

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

CHAPTER IV

# Background

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

## Time Series

A time series is a collection of observations that occur in order. The process underlying a time series is assumed to be stochastic, so the model must correspondingly be probabilistic. Critically, the sequence of observations cannot be re-arranged, as each observation is typically dependent on one or more previous observation. This dependence is termed autocorrelation and accounting for it is one of the primary functions of a time series model.

## Autoregressive Models

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

where , , …, are the parameters, is a constant, and is the white noise term.

When the AR model is extended to the multivariate case (i.e. allowing for multiple time series), a Vector AR (VAR) model is formed. This model will support not only a time series for defect count, but also time series for the two release plan variables: improvements and new features.

## Endogeneity and Exogeneity

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

The alternative is that a time series should not be explained by itself, and is only used to explain other time series. This type of explanatory variable is called exogenous, and could be considered an input.

By also considering exogenous variables, a VAR model would become a VARX model. This model meets the requirements of the explanatory model described in the Motivation section, since it would allow release plan variables to be kept exogenous and used only to explain defect count.

## Trends and Stationarity

AR, VAR, and VARX models do not account for non-stationary data. If a time series is not stationary, differencing may produce a stationary series. Trending time series are challenging to analyze, because the summary statistics of mean, variance, and autocovariance vary over time, and are therefore not interpretable [5]. Two trend types are discussed here: deterministic and stochastic.

A deterministic trend will move upward or downward, meaning that the time series mean is non-constant. However, the time series will be constant according to a deterministic function and the time series movements will generally follow the deterministic function, with non-permanent fluctuations above or below. Such a time series is said to be stationary around a deterministic trend.

In contrast, a stochastic trend shows permanent effects whenever random variations occur, and the series will not necessarily fluctuate only close to the area of a deterministic function. The application of differencing can be used to remove a stochastic trend.

Stationarity can be strict or weak (of some order). Strict stationarity occurs when statistical properties are invariant with respect to shifts of the time origin [12]. Alternatively, a weak stationarity (of second order) can be established, and from this strict stationarity can be established by then assuming normality [4].

For a multivariate time series, stationarity holds if all the component univariate time series are stationary [16], so the goal of stationarity testing will be to establish second-order stationarity for each univariate time series component, and then show that the assumption of normality is reasonable. This will establish the stationarity of the multivariate time series as a whole. Next, tests are discussed for assessing if a deterministic or stochastic trend is present.

## Unit Root and Stationarity Testing

A time series that contains a stochastic trend is non-stationary. A pure auto-regressive (AR) model of such a time series contains a unit root [5]. Testing for the presence of a unit root can therefore be used to test for non-stationarity. A unit-root test poses as the null hypothesis that an AR model has a unit root. Then, a test statistic is measured. If the p-value is below some significance, the null hypothesis can be rejected, and it is established that the time series does not have a stochastic trend. The Augmented Dickey Fuller (ADF) test is often used for unit root testing.

On the other hand, a stationarity test uses the null hypothesis that a time series is stationary around a deterministic trend. If the test statistic shows that this hypothesis can be rejected, at some significance level, then a stochastic trend should be considered, by the unit root test. The Kwiatkowski–Phillips–Schmidt–Shin (KPSS) test can be applied for testing stationarity.

CHAPTER V

# Methods

In this chapter, we consider methods for both obtaining time series data (data methods) and for obtaining a model using that data (modeling methods).

## Data Methods

In this section, the data sources and the rationale for their selection are discussed. Then the methods used for preparing data for modeling, by cleansing, sampling, stationarity testing, and windowing, are described.

### Data Sources

The empirical datasets used to establish predictive models came from several software projects’ historical data, and were taken from their issue tracking systems[[1]](#footnote-1). To be considered for selection, it was required that a project that

1. Has been actively developed for at least several years
2. Has openly available issue tracking system data
3. Distinguishes between defects and other issue types

The projects selected by these criteria were:

* MongoDB*[[2]](#footnote-2)*: *core server* product
* Hibernate[[3]](#footnote-3): *orm* product
* NetBeans[[4]](#footnote-4): *platform* and *java* products

The MongoDB software project has been actively developed since 2009. MongoDB uses JIRA[[5]](#footnote-5) for issue tracking. Issue data for *core server* product was exported from the project's JIRA web interface[[6]](#footnote-6) as XML data.

The Hibernate software project has been actively developed since 2003, and also uses JIRA for issue tracking. Issue data for the *orm* product was exported from the project’s JIRA web interface[[7]](#footnote-7) as XML.

The Netbeans software project has been actively developed as an open source project since 2000. The project uses Bugzilla for issue tracking. Issue data for the *platform* and *java* products was obtained using a 2010 dump of the Bugzilla MySQL database. This database was made available as part of the mining challenge for the 2011 conference for Mining Software Repositories[[8]](#footnote-8).

### Data Preparation

The raw software issue data needs preparing before a time series modeling procedure is run. Preparatory steps include: cleansing, sampling, stationarity testing and differencing, and windowing. These steps are now explained below.

#### Data Cleansing

Not all of the data was preserved for modeling. The modification or removal of data is discussed next. Then the steps of sampling and windowing are discussed.

First, only issues with resolutions such as *fixed*, *complete*, or *done* will be kept. Issues with other resolutions, such as *unresolved*, *won't fix*, *duplicate*, etc. were counted as unfixed and were not kept. This was done because the proposed model structure assumes that bug creation is explained by software changes. Therefore, issues that do not result in any change were not included in the dataset.

Next, issues that are categorized as sub-tasks are converted to be the same issue type as the parent issue. Those sub-tasks whose parent issue is not in the dataset are considered orphans and discarded.

#### Data Sampling

Data was sampled at regular periods to measure the following: number of improvements resolved, number of features resolved, and number of bugs created. As an example, this sampling process is illustrated in Figure 3, with the outcome of sampling the example data shown in Table 1.

#### Establishing Stationarity

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

The *urca[[9]](#footnote-9)* library provides ur.df and ur.kpss functions for performing these tests.



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

Table Results of sampling example issues shown in Figure 3.

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

#### Time Windowing

It is assumed that the software development process underlying a given project might change over time. Rather than developing a model that also changes over time, the data will be kept for modeling only if it occurs within a time window. This will limit the amount of process change the model is exposed to. Taking this approach means that the entire modeling methodology will be executed for each time-windowed part of the data. See an illustration of a window in Figure 4.

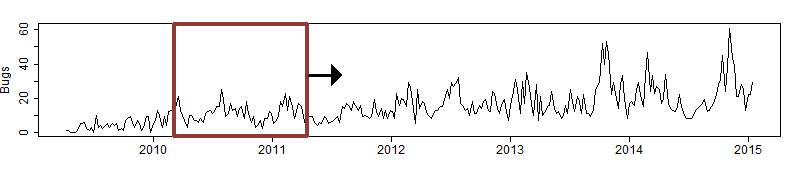


Figure An illustration of time-windowing, where only data within the window is used for modeling.

It will be necessary to advance the time window after modeling the data within the window, so that the entire time series can take part in the modeling. This notion of applying modeling data within the window, advancing the window by one sample, and then repeating until the end of the time series is reach, is called herein a sliding window.

## Modeling Methods

The typical methodology used for building time series models involves specification, estimation, and diagnostics checking [4]. Once specified and estimated, the diagnostic checking step ensures that only valid models are considered for selection. The final step of modeling would be selection, where models are compared by some model selection criterion [4, pg. 581]. This section presents the approach used to specify, estimate, check, and select a VARX model to be used for defect prediction.

### Model Specification & Estimation

Specification of a 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 time series variables, there are unknown parameters to be estimated. Let the ratio of observations to parameters be denoted by

To keep at or above some minimum ratio , so there are not too few observations per parameter, we form the inequality

In terms of this becomes

Then, 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 1, 2,..., . 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[[10]](#footnote-10)* library provides the estVARXar function.

### Diagnostics Checking

Diagnostic checking is performed to verify that a model can be accepted. This step includes testing for stability and for model inadequacy.

For model with an AR portion to be stable, the roots of the process characteristic equation must lie outside the unit circle [4, p. 56]. Equivalently, the inverse of the roots must lie inside the unit circle.

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.

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[[11]](#footnote-11)* library provides the Box.test function for performing the Ljung-Box test.

### Model Selection

Model selection criteria are used to compare models according to their fit, by penalizing for residual error and the number of parameters. There are a number of different selection criteria, including Akaike Information Criterion (AIC), AIC with correction (AICc), and Bayesian Information Criterion (BIC). Bisgaard and Kulahci noted that “. . .[t]he penalty for introducing unnecessary parameters is more severe for BIC and AICC than for AIC” [3]. A less severe penalty for the number of parameters would be preferred in this case, since we are already limiting the number of parameters in the model specification step, and because additional parameters may in fact be necessary to account for time series autocorrelations with higher lags. Therefore, AIC was chosen as the selection criterion.

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

CHAPTER VI

# Results

The data and modeling methods described in the Methods chapter were applied to the four datasets: MongoDB *core server*, Hibernate *orm*, NetBeans *platform* and *java*. The results of applying the methods are described in the following sections.

## Data Results

Data was collected from project issue tracking systems. Table 2 shows the range of dates over which data was collected for each project product, and the number of issues that were collected as a result. This issue count does not include issues that were excluded as part of data cleansing (see the Data Cleansing section).

Table Date ranges of data collected, and the number issues that resulted.

|  |  |  |
| --- | --- | --- |
| Project Product Name | Date Range | Issue Count |
| MongoDB *core server* | Apr, 2009 – Jan, 2015 | 7,042 |
| Hibernate *orm* | Apr, 2003 – Apr, 2015 | 8,315 |
| NetBeans *platform* | Jan, 2001 – Jun, 2010 | 11,362 |
| NetBeans *java* | Jan, 2001 – Jun, 2010 | 8,734 |

### Sampling Results

The collected datasets were then sampled to create time series. Sampling was performed for each of the following sampling periods: 7 days, 14 days, and 30 days. For the 30-day sampling period, the resulting time series are shown in Figure 5 through Figure 8.



Figure Time series resulting from sampling the MongoDB *core server* dataset with a 30-day sampling period.



Figure Time series resulting from sampling the Hibernate *orm* dataset with a 30-day sampling period.



Figure Time series resulting from sampling the NetBeans *platform* dataset with a 30-day sampling period.



Figure Time series resulting from sampling the NetBeans *java* dataset with a 30-day sampling period.

### Stationarity Testing Results

The resulting time series were then tested for stationarity. The time series we found to be non-stationary, with the exception of the Hibernate *orm* dataset, which was stationary when using a 30-day sampling period. Differencing was found to remove non-stationarity.

### Windowing Results

For each sampling period, a range of window sizes were selected to be used for the sliding window, as shown in Table 3.

Table Sliding windows sizes to be used for each sampling period

|  |  |
| --- | --- |
| Sampling Period | Sliding Window Sizes |
| 7 days | 36, 39, 42, 45, 48, 51, 54, 57, 60, 63, 66, 69, 72, 75, 78 |
| 14 days | 24, 27, 30, 33, 36, 39, 42, 45, 48, 51, 54 |
| 30 days | 12, 15, 18, 21, 24, 27, 30, 33, 36 |

## Modeling Results

CHAPTER VII

# Conclusions & Future Work

# Rerferences

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. An issue tracking system can be used to track bugs, new features, improvements, etc. [↑](#footnote-ref-1)
2. MongoDB is a scalable document-oriented database system (http://www.mongodb.org/). [↑](#footnote-ref-2)
3. Hibernate is an object-relational mapping (ORM) framework for the Java language. [↑](#footnote-ref-3)
4. NetBeans is a software development platform written in Java [↑](#footnote-ref-4)
5. JIRA is an issue tracking and project management system made by Atlassian [↑](#footnote-ref-5)
6. The project’s JIRA web interface is at https://jira.mongodb.org/browse/SERVER [↑](#footnote-ref-6)
7. The project’s JIRA web interface is at https://hibernate.atlassian.net/projects/HHH [↑](#footnote-ref-7)
8. The mining challenge data is available at http://2011.msrconf.org/msr-challenge.html [↑](#footnote-ref-8)
9. The *urca* library (http://cran.r-project.org/web/packages/urca) provides tests for time series data, and is freely available as a package for the *R* computing environment. [↑](#footnote-ref-9)
10. The *dse* (http://cran.r-project.org/web/packages/dse) library provides tools for time series models, and is freely available as a package for the *R* computing environment. [↑](#footnote-ref-10)
11. The *stats* (http://stat.ethz.ch/R-manual/R-patched/library/stats/html/00Index.html) library provides core statistics functions, and is freely available as a package for the *R* computing environment. [↑](#footnote-ref-11)