

Software Defect Data and Predictability for Testing Schedules

Rattikorn Hewett & Aniruddha Kulkarni
Dept. of Comp. Sc., Texas Tech University
rattikorn.hewett@ttu.edu
aniruddha.kulkarni@ttu.edu

Catherine Stringfellow
Dept. of Comp. Sc.
Midwestern State University
catherine.stringfellow@mwsu.edu

Anneliese Andrews
Dept. of Comp. Sc.
University of Denver
andrews@cs.du.edu

Abstract

Software defect data are typically used in reliability modeling to predict the remaining number of defects in order to assess software quality and release decisions. However, in practice such decisions are often constrained by availability of resources. As software gets more complex, testing and fixing defects become difficult to schedule. This paper attempts to predict an estimated time for fixing software defects found during testing processes. We present an empirical approach that employs well-established data mining algorithms to construct predictive models from historical defect data. We evaluate the approach using a dataset of defect reports obtained from testing of a release of a large medical system. The accuracy obtained from our predictive models is as high as 93% despite the fact that not all relevant information was collected. The paper discusses detailed methods of experiments, results and their interpretations.

1. Introduction

Software testing requires rigorous efforts, which can be costly and time consuming. It can easily take 50% of a project life cycle. As software projects get larger and more complex, testing and fixing defects become difficult to schedule. Making decisions on suitable time for testing termination and software release often requires considerations of tradeoffs between cost, time and quality.

Software defect data have been used in reliability modeling for predicting the remaining number of defects in order to assess quality of software or to determine when to stop testing and release the software under test. Testing stops when reliability meets a certain requirement (i.e., number of defects is acceptably low), or the benefit from continuing testing cannot justify the testing cost. While this approach is useful, it has some limitations. First, decisions on testing activities involve three aspects of software testing processes: cost, time and quality but reliability is mainly concerned with the “quality” aspect. Second, although the number of defects is directly related to the time required for fixing the defects found, they do not necessarily scale in linear fashion. A large number of simple defects may take much less time to fix than a few sophisticated defects. Similarly, the time required to fix the defects could also depend on the experience and skills

of the fixer. Finally, defect reports are often under utilized. Existing approaches to reliability modeling tend to exploit only quantitative measures. However, there are many factors, other than the number of defects, which could influence the time to fix the bugs. Many of these factors are qualitative. During the testing process, data about defects are documented in a software defect (bug) reports. These reports collect useful historical data about software defects including locations and types of defects, when they were found and fixed, and names of the testing and fixing teams. An approach to analysis that can utilize both quantitative and qualitative data in the defect report would be useful.

This paper proposes a novel approach to use software defect data for predicting a “time” aspect as opposed to the “quality” aspect of software testing process. In particular, we propose an approach to create a predictive model, from historical data of software defects, for predicting an estimated time to fix the defects during software testing. Such predictions are useful for scheduling testing and for avoiding overruns in software projects due to overly optimistic schedules.

Estimation of the time required to fix the defects is a difficult problem. We have to understand more about defects to be able to make predictions about fixing them. Causes of software defects include design flaws, implementation errors, ambiguous requirements, requirements change, etc. Defects can have varying degrees of complexity (which is proportional to the amount of time required to fix the defect) and severity (the extent to which the defect impairs the use of software). Furthermore, defects can be found by various groups and in various phases in the life cycle of software development. Defects found by the system testing group may be harder to fix than those found in an earlier stage of software life cycle. In some cases errors encountered by users may be difficult to reproduce. To make the matter worse, predicting time to fix defects is hard, as it is possible that fixing one problem may introduce a new problem into the system

This research proposes an empirical study to investigate the above problem by means of advances in data mining. Our main contributions include:

- (1) formulation of a challenging new problem and approaches for the solution that are potentially useful

for scheduling and managing the software testing process

- (2) empirical approaches to increase utilization of defect data by exploiting both quantitative and qualitative data factors

The rest of the paper is organized as follows. Section 2 discusses related work. Section 3 describes the defect data set followed by the details of the proposed approach in Section 4. Section 5 evaluates the proposed approach by experimentation on a real-world data set and comparing accuracy of results obtained from various predictive models using different data mining algorithms. The paper concludes in Section 6.

2. Related Work

To achieve effective management in testing process, much work has been done on using mathematical models for estimating “testing efforts,” including the widely used method COCOMO [2]. However, testing efforts often consider only resources required for testing design and execution, and do not include the resources spent on fixing defects themselves [3]. Thus, estimating testing efforts is very different from our work, which involves estimating the time required for fixing defects.

Most of the work related to the utilization of software defect data has been in determining fault-prone components [1, 9, 14, 15] and in reliability modeling to predict the reliability of the software (the remaining number of defects) [5, 11, 12]. Reliability modeling makes limited use of the data as it only uses the number of defects found during different testing periods. Biyani and Santhanam [1] illustrated how defect data can be used to compare release qualities and relation between number of defects pre-release and post-release. In addition, empirical study in [9] uses the data to construct models to classify quality of software modules during software testing. Our work offers a new way in which one can make use of defect data. Furthermore, our models can incorporate both quantitative and qualitative values.

3. Data

To evaluate our approach, we experiment with a defect data set obtained from testing of three releases of a large medical record system as reported in [13]. The system initially contained 173 software components and 15 components were added over the three releases. Each data instance represents a defect with various corresponding attribute values including the defect report number, the release number, the phase in which the defect occurred (e.g., development, test, post-release), test site reporting the defect, the component and the report date.

| Name | Description | Data Types |
|---------------|---|--|
| Prefix | The development group, which found the defect. | 8 discrete values, e.g., development, system testing |
| Answer | Response from fixer indicating the type of the defect | 17 discrete values, e.g., limitation, program defect, new function |
| Component | The component to which the defect belongs | 75 discrete values |
| State | Status of the defect | 5 discrete values, e.g., canceled, closed, verify |
| Severity | The severity of the defect | 4 values: 1, 2, 3, 4 (1 = low, 4 = high) |
| OriginID | Person who found the defect | 70 discrete values |
| OwnerID | Person who is fixing the defect | 57 discrete values |
| AddDate | Date on which defect was added | Numeric |
| AssignDate | Date on which it was assigned to someone to work on | Numeric |
| Response Date | Date on which the fixer responded with an “answer” (attribute answer) | Numeric |
| EndDate | Date on which the defect was closed | Numeric |
| LastUpdate | This is the date on which the defect was last updated | Numeric |

Table. 1 Data Characteristics.

For the study in this paper, we use data from release 1, which contains 1460 defects (or data instance or rows). There are a total of 30 attributes, 12 of which are selected to be relevant to our problem. Table 1 summarizes the attribute descriptions and their corresponding data types.

4. Approach

4.1 Problem formulation

Unlike most work in software quality, in this study we propose to predict the time required for fixing defects. For clarity, we describe relevant terms in our defect report related to different stages of a defect life cycle as summarized in a timeline in Figure 1. These terms are defined below in more details.

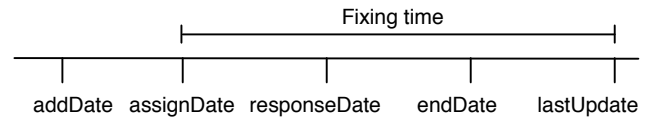


Fig. 1 Timeline for different dates in the data.

AddDate is the date on which the defect was found and added in the defect report. *AssignDate* represents the date on which a defect was assigned to be fixed. Because we are concerned with the time spent on fixing defects we can safely ignore the time spent in between addDate and assignDate. *ResponseDate* is the date on which the person

assigned to fix the defect determines the suitable course of action with respect to the defect. *EndDate* specifies the end of the major fixing period that starts from the *ResponseDate*. *LastUpdate* signifies the end of the fixing process with respect to a particular defect. After the major fixing period if anymore problems are discovered, the developer fixes them and updates the *endDate* to the *lastUpdate* attribute.

A person assigned to fix a defect spends the time from *assignDate* till *lastUpdate* in fixing the defect. Our interest is to predict the time period from *assignDate* to *lastUpdate* as opposed to the period from *addDate* to *lastUpdate*. We ignore the period between *addDate* and *assignDate*, which is the time spent on dealing with management issues rather than the time that is actually spent on fixing the defect. Thus, our problem is to construct a predictive model that best fits a defect data set. The data set has a class attribute referred to as *fixing-time*, which represents a number of days required for fixing a particular defect until release. The fixing-time is computed by subtracting *assignDate* from *lastUpdate*. Because the *responseDate* and the *endDate* occur during the period that we want to predict, knowing their values would certainly affect the predicted value and consequently makes the prediction easier. Thus, using *responseDate* and *endDate* as condition attributes for predicting the fixing-time would be tantamount to cheating, as the predicted values are part of these values that are known prior to the prediction. As a result, for model prediction, in addition to the class attribute we employ all but the last three of selected relevant attributes (as shown in Table 1) in our analysis.

4.2 Relevant analysis issues

One common issue in data analysis in software domains is that the data is very scarce or when it is available it tends to be incomplete. Our problem is no exception. There are a number of factors that influence the time required to fix software defects including number of lines of code or modules changed or added, complexity of algorithms or logical controls or interactions, skills of fixers and testers, etc. Unfortunately, not all information about these factors is available in the defect report.

Another issue concerns the data value types, which include both quantitative and qualitative data. Most existing modeling approaches (e.g., statistical or time series techniques) employ quantitative models, where all variables have continuous or discrete values. However, many of the relevant factors in the defect report can be qualitative (e.g., names of person who fix the defect, component where the defect was found, etc.) A common approach to deal with this problem is to convert qualitative values into quantitative values. While this is useful in some cases, it imposes an order constraint on

values and not all categorical values are ordinal. It is desirable to have an analysis technique that can directly analyze qualitative (or symbolic) as well as quantitative data.

Advanced research in data mining has produced many data analysis techniques that can analyze both quantitative and qualitative data including association rule mining, Naïve Bayes classification technique and decision tree learning [7, 10, 12, 16]. The decision tree learner is one of the most prominent machine learning techniques that been applied successfully in classification problems, where a class attribute has discrete value. Thus, viewing our problem as a classification problem, we need to have discrete class attribute values in order to be able to apply these modeling techniques (e.g., decision tree learning and Naïve Bayes technique). In particular, we need to discretize the class attribute, fixing-time. In this study, we use equal frequency bins to discretize the class fixing-time. The resulting values are in three categories: 0-59 days, 60-103 days and more than 103 days. This method reflects natural clusters of the data set and appears to be meaningful in real software practices. The large grain size of this fixing-time reflects the fact that in real practices a fixer may have to fix multiple defects in the same time period.

4.3 Approaches to data analysis

In this study, we employ four supervised learning algorithms based on three different approaches for constructing predictive models from a data set. We select these approaches since each is based on a different model representation and learning method as described below.

Decision tree learner [12] is one of the most popular methods in data mining. A decision tree describes a tree structure wherein leaves represent classifications (or predictions) and branches represent conjunctions of conditions that lead to those classifications. To construct a decision tree, the algorithm splits the data source set into subsets based on an attribute value of the splitting attribute (selected by a rank-based measure called ratio gain). This process repeats on each derived subset in a recursive manner until either splitting is not possible, or a singular classification is applied to each element of the derived subset. See more details for decision table learners in [10].

Naïve Bayes Classifier [7] is based on probability models that incorporate strong independence assumptions that often have no bearing in reality, hence are “naïve”. The probability model can be derived using Bayes’ theorem. Bayesian classifiers are called active learning, as they require little or no training time. The classification is computed based on the likelihood for each attribute belonging to a particular class. See more details in [7].

Neural net approach is based on a neural network, which is a set of connected input/output units (perceptrons) where each connection has a weight associated with it [8]. During the learning phase the network learns by adjusting the weights so as to be able to predict correct target values. Back propagation is one of the most popular neural net learning methods. The net learns by iteratively processing a set of training samples and comparing the networks prediction for each sample with the actual known class label. For each training sample, the weights are modified so as to minimize error between the predicted and the actual values. The weight modifications are made in backward direction that is from output layer to input layer, hence the name back propagation. The neural net approach generally takes a long time to train and requires parameters that are best determined empirically. They do, however, have a high tolerance to noisy data as well as the ability to fit complex (non-linear) patterns.

5. Experiments and Results

To validate our proposed idea described in Section 4, we perform experiments to create and evaluate predictive models for predicting estimated time to fix defects.

5.1 Data preprocessing

Our experiments involve three types of data preprocessing: *data selection*, *data conversion*, and *data discretization*.

Because we are interested in estimating the time to fix defects, we select only data points representing the defects that are caused by malfunctions or faulty components as opposed to those that are reported as defects due to misuse of test cases, fixing postponement or software limitation. In other words we consider only defects that are identified by fixers and are completely fixed (i.e., state attribute value is closed). After the data selection our data set remains with 1357 data points.

In data conversion, several attributes with date values are converted into numbers by subtracting all the dates from January 1, 1995. We chose this date partly because all the dates in the data start from 1996, and thus the resulting numbers are all positive, which are easier to deal with and to interpret. Interestingly, Microsoft Excel by default converts dates into numbers by subtracting the dates by January 1, 1900. If we used this default setting, the model accuracy obtained would not have been as accurate as the cutoff date we proposed. This is because the resulting numbers using the default cutoff date are so large that they reduce the power to differentiate different dates and their impacts on the fixing-time.

The final step is data discretization. Our defect data set contains both continuous and nominal values. Certain analysis algorithms (e.g., Naïve Bayes classifier) require

continuous attribute values to be discretized. In this study, we apply two discretization methods: the *equal frequency bins* and the *entropy-based discretization* [4]. Unlike the binning approach, the entropy-based discretization uses class information for discretization and has shown to perform well [7]. For comparison purpose, we will also create models that do not require discretization.

5.2 Experimentation

We apply four data mining algorithms: *NaiveBayes*, *MultilayerPerceptron*, *DecisionTable* and *J48* (a variation of C4.5 [12]), provided by the data mining tool WEKA [16], as representative systems for the Naïve Bayes classifier, the Neural Net approach, and the decision tree learners, respectively. The *MultilayerPerceptron* is based on a back propagation learning algorithm described earlier. For our experiments we use a default setup of a network with two hidden layers, a learning rate of 0.3, a momentum of 0.2 and 30 iteration cycles as a bound for termination. The network configuration and parameters are obtained by using standard empirical procedures. See more details on relevant parameters and their meaning in [16].

To avoid overfitting, *n-fold cross-validation*, a standard re-sampling accuracy estimation technique, is used [10]. In *n-fold* cross validation, a data set is randomly partitioned into *n* approximately equally sized subsets (or folds or tests). The learning algorithm is executed *n* times; each time it is trained on the data that is outside one of the subsets and the generated classifier is tested on that subset. The estimated accuracy for each cross-validation test is a random variable that depends on the random partitioning of the data. The estimated *accuracy* (i.e., ratio of a number of correct predictions to a total number of test cases) obtained from 10-fold cross validation is computed as the average accuracy over the *n* test sets. The *n-fold* cross-validations are typically repeated several times to assure data randomness; and the estimated accuracy is an average over these *n-fold* cross-validations. For the experiments in this paper, the accuracy result is an average accuracy obtained when *n* = 10, as suggested in [10].

5.3 Attribute Selection

Although our set of relevant attributes is not extremely large, we apply a standard technique for attribute selection to maximize possible accuracy obtained. Our attribute selection technique is based on a *wrapper method* [6]. The method can be viewed as a greedy search to find a state associated with the subset of attributes that gives the highest heuristic evaluation function value (in hill-climbing, depth first search fashion). The heuristic function value here is the estimated future accuracy obtained from an average

accuracy from predictive models obtained from a 10-fold cross validation on the attribute set of the next state. The search starts from a state with an empty set of attributes, repeatedly moves to a state containing a larger set of attributes, and stops after a fixed number of node expansions does not yield a future state with a higher estimated accuracy than those of the current state.

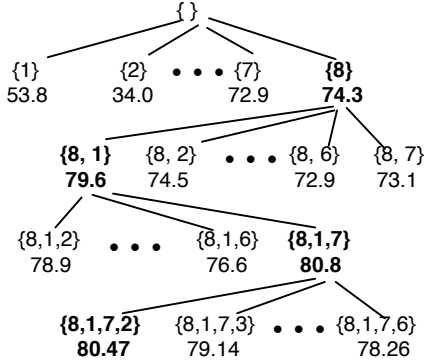


Fig. 2 Greedy search for the “best” set of attributes.

Figure 2 shows a partial search state space resulting from the wrapper method using the average accuracy obtained from decision tree models as the heuristic evaluation function. We apply the method to a total of eight relevant condition attributes (the top nine attributes of Table 1, according to the data selection, with an omission of the state attribute). Each state represents a set of attributes, each of which is represented by a number corresponding to its order in Table 1 (e.g., 1, 7 and 8 represent the attributes Prefix, AddDate and AssignDate, respectively). The best set at each search level is shown in bold. As shown in Figure 2, the optimal set of attributes obtained is {8, 1, 7}. We obtain the same set of attributes using predictive models learned from Naïve Bayes classifier.

5.4 Results

Table 2 shows average percentages of accuracy (over 10-fold cross validation) obtained from four different types of data analysis algorithms: Naive Bayes classifier, Decision Tree learner, decision table learner, and neural net approach. The ZeroR learner predicts values based on the majority of the class distribution and is commonly used as a measure of baseline performance. Table 2 compares results obtained with (bottom part of the table) and without (top part of the table) attribute selection using two different techniques for discretizing continuous attribute values. With an exception of the “Neural Net (cont. class)”, all results are obtained using the data set with discrete class attribute values as discussed in Section 4.2. To give a fair assessment of a learning approach that

can predict a continuous value (e.g., neural net approach), we include the results obtained by using the data set with continuous class values as well (e.g., the “Neural Net (cont. class)” shown in Table 2). The accuracy in such case is based on the number of correct predictions using the same range of the discrete intervals (i.e., 0-59 days, 60-103 days, more than 103 days) for original value and predicted value in order to compare with other learning approaches. Note that the entropy-based discretization requires class information (or labeled data), which is not available for the “Neural Net (cont. class)” case.

| No attribute selection | Discretization Method | | |
|--------------------------|-----------------------|--------------|---------------|
| Learning Approach | None | Binning | Entropy-based |
| NaiveBayes | 77.22 | 74.42 | 79.14 |
| Decision Tree | 93.51 | 78.11 | 92.33 |
| Decision Table | 92.4 | 78.18 | 92.33 |
| Neural Net | 33.33 | 55.63 | 59.32 |
| Neural Net (cont. class) | 58.8 | 65.58 | na |
| ZeroR | 33.16 | 33.16 | 33.16 |

| With attribute selection | Discretization Method | | |
|--------------------------|-----------------------|---------|---------------|
| Learning Approach | None | Binning | Entropy-based |
| NaiveBayes | 80.84 | 76.78 | 80.91 |
| Decision Tree | 93.44 | 77.89 | 91.96 |
| Decision Table | 92.4 | 78.18 | 92.33 |
| Neural Net | 85.7 | 78.85 | 89.16 |
| Neural Net (cont. class) | 83.79 | 67.13 | na |
| ZeroR | 33.16 | 33.16 | 33.16 |

Table. 2 Average accuracy of predictions.

As shown in Table 2, with or without attribute selection, Naïve Bayes and neural net techniques perform best when entropy-based discretization is used. However, neural net with continuous class value has the best accuracy when using no discretization or equal frequency binning. In general, the results obtained with or without attribute selection are consistent except for those obtained from the neural nets approach, where the accuracy increases by close to 30% when a selected set of attributes is used. In fact, the model obtained by the decision tree algorithm has the highest accuracy of 93.5% followed by 92.4 % accuracy of the decision table model. Both of the top resulting models when all attributes are used are almost the same as those obtained when a selected set of attributes is used. Both use no discretization. However, the results obtained from decision tree and decision table learners using the entropy-based discretization are no more than 1.2 % less than the best two accuracies. Even though the accuracy obtained from the neural net model increases when we use a selected set of attributes, it still lags behind the top two approaches by about 3%. Using a selected set of attributes, the neural net approach with discrete class outperforms that with continuous class values. The overall results obtained from various algorithms are far better than an accuracy obtained from a random guess of 33.3% as indicated by the ZeroR approach in Table 2.

| | |
|---------|---|
| Rule 1. | $\text{assignDate} \leq 909 \Rightarrow (\text{fixing-time} > 103)$: 381/2. |
| Rule 2. | $909 < \text{assignDate} \leq 951$ and $\text{prefix} = \text{kt}$ and $\text{addDate} \leq 944.5 \Rightarrow (\text{fixing-time} > 103)$: 12/0. |
| Rule 3. | $\text{assignDate} > 951$ and $\text{prefix} = \text{bt} \Rightarrow (\text{fixing-time} \leq 59)$: 141/16. |
| Rule 4. | $909 < \text{assignDate} \leq 998$ and $\text{prefix} = \text{kt} \Rightarrow (59 < \text{fixing-time} \leq 103)$: 143/17. |
| Rule 5. | $\text{assignDate} > 951$ and $\text{prefix} = \text{bd} \Rightarrow (\text{fixing-time} \leq 59)$: 116/9. |

Fig. 3 Example rules from a predictive model.

Figure 3 illustrates examples of the rules extracted from a decision tree model constructed from overall 1357 training instances. Each rule represents a path from the root to a leaf. Our resulting model is a tree of size 31 with a total of 23 rules. The model has 93.5% accuracy on the training set and has a root mean squared error of 0.203. As shown in Figure 3, the end of each rule follows by “: x/y ”, where x and y represent a number of training instances (with the same condition as the rule condition) that are correctly and incorrectly predicted by the rule, respectively. Recall that each date is converted into number of days in a period between the date and a fixed cutoff date (January 1, 1995). Thus, if the assignDate is “small” then the defect was assigned “early” for fixing. For example, Rule 1 gives an implication that if the defect has been assigned “early,” the fixing-time is likely to take longer than 103 days (with support evidences of about 28%). In practice, this may be the case that the fixer delays fixing defects as the deadline is not close or that the type of defects found early may be due to missing functionality that requires time to implement. Rule 2 suggests that if the defects were not assigned “late” and were found by a customer testing group, then the fixing-time is likely to be long. Rule 3 reflects the situation when the defect was assigned “late” and found by a system testing group then the fixing-time is likely to be short, i.e., no more than 59 days. The type of defects is not likely to be a major functional defect and therefore is likely to take less time to fix. Similarly, Rule 5 is concerned with the defect found by the developers. Thus, it is likely to be found in an early stage, which takes short time to fix. As shown in Figure 3, Rule 2 is the weakest of the five rules in terms of support degree as evidenced by the number of matching data instances.

6. Conclusion

We present a novel approach to utilize software defect data to create predictive models for forecasting the estimated time required for fixing defects by means of advances in data mining. Such estimation has potential benefits for planning and scheduling in software testing management. Although approaches to estimating testing

efforts exist, estimation of time required for testing and fixing problems are two different problems. We illustrate an approach to the solution of the latter. The results of our predictive models obtained from various data mining algorithms are promising with an average of the best to be over 90%. However, like any other modeling approach that is based on historical data, our approach has some inherent limitations in that the resulting models can only be applied to software projects that share the same characteristics (i.e., schema or a set of condition attributes and the attribute domain). One remedy for this issue is to build predictive models from a set of defect data sets collected from different class of projects (e.g., organic, embedded as in COCOMO [2]). This should extend the generality of the model application.

References

- [1] Biyani, S., P. Santhanam, “Exploring Defect Data from Development and Customer Usage on Software Modules over Multiple Releases,” in *Procs of Int'l Conf. on Software Reliability Eng.*, Paderborn, Germany, pp. 316–320, 1998.
- [2] Boehm, B., *Software Engineering Economics*, Prentice Hall, 1981.
- [3] Culbertson, R., C. Brown and G. Cobb, *Rapid Testing*, Prentice Hall, Dec 29, 2001.
- [4] Usama M. Fayyad and Keki B. Irani, “Multi-interval discretization of continuous valued attributes for classification learning,” in *Proceedings of IJCAI-93, volume 2*, Morgan Kaufmann Pub., pp. 1022-1027, 1993.
- [5] N. E. Fenton, and M. Neil, “A critique of software defect prediction models,” *IEEE Trans. Software Engineering*, 25(5): pp. 675–689, 1999.
- [6] John, G.H., Kohavi, R., Pflieger, K., Irrelevant Features and the Subset Selection Problem, Proc. of the 11th International Conference on Machine Learning ICML94, pp. 121-129, 1994.
- [7] Jiawei Han, and Micheline Kamber, *Data Mining: Concepts and Techniques*, Morgan Kaufmann, 2000.
- [8] Simon Haykin, *Neural Networks: A Comprehensive Foundation* (2nd Edition), Springer, Springer-Verlag New York Inc June 1, 1995,
- [9] Khoshgoftaar, T., R. Szabo, , T. Woodcock, , “an Empirical study of Program Quality During Testing and Maintenance,” *Software Quality Journal*, 137-151, 1994.
- [10] Kohavi, R., “The Power of Decision Tables,” In European Conference on Machine Learning, Springer Verlag, 1995.
- [11] Musa, J., A. Iannino and K. Okumoto, *Software reliability: measurement, prediction, application*, McGraw-Hill, Inc., New York, NY, 1987.
- [12] Quinlan, R., *C4.5: Programs for Machine Learning*, Morgan Kaufmann Publishers, San Mateo, CA, 1993.
- [13] Stringfellow, C. and A. Andrews, “An empirical method for selecting software reliability growth models,” *Empirical Software Engineering*, 7(4): pp. 319–343, 2002.
- [14] Stringfellow, C. and A. von Mayhauser, “Deriving a Fault Architecture to Guide Testing,” *Software Quality Journal*, 10(4), December, 2002, pp. 299-330.
- [15] Stringfellow, C. and A. Anfrews, “Quantitative Analysis of Development Defects to Guide Testing,” *Software Quality Journal*, 9(3), November 2001, pp. 195-214.
- [16] Witten, I. and E. Frank, *Data Mining practical Machine Learning Tools and Techniques*, Morgan Kaufmann, San Francisco, CA, 2005.