

On modeling software defect repair time

Rattikorn Hewett · Phongphun Kijsanayothin

Published online: 6 May 2008

© Springer Science + Business Media, LLC 2008

Editor: Tim Menzies

Abstract The ability to predict the time required to repair software defects is important for both software quality management and maintenance. Estimated repair times can be used to improve the reliability and time-to-market of software under development. This paper presents an empirical approach to predicting defect repair times by constructing models that use well-established machine learning algorithms and defect data from past software defect reports. We describe, as a case study, the analysis of defect reports collected during the development of a large medical software system. Our predictive models give accuracies as high as 93.44%, despite the limitations of the available data. We present the proposed methodology along with detailed experimental results, which include comparisons with other analytical modeling approaches.

Keywords Quality assurance · Software testing · Testing management · Defect report analysis · Data mining

1 Introduction

Software quality has a profound impact on overall system performance. Producing high quality software requires minimizing the number of defects in the final product, and this depends on testing quality and management. Software testing is challenging, costly and time-consuming. It can easily take 50% of development time (Boehm and Basili 2001). Greater accuracy in predicting the time needed to repair defects is useful for devising better testing plans, schedules, and for allocation of testing resources. Effective testing management can significantly enhance the quality of software and helps in keeping projects on time and within budget. Current practices for managing defects, planning testing strategies and rationally allocating test resources are not well understood (Galen 2005).

R. Hewett (✉) · P. Kijsanayothin
Department of Computer Science, Texas Tech University, 302 Pine Street, Abilene, TX 79601, USA
e-mail: Rattikorn.Hewett@ttu.edu
URL: <http://www.cs.ttu.edu/~hewett>

P. Kijsanayothin
e-mail: pkijšana@cs.ttu.edu

Although research has produced advances in software technology, the underlying theoretical principles of software development remain difficult to follow in practice. Due to technologies such as defect tracking tools, many software developers have data collection mechanisms in place and software development data has become more readily available. As a result, empirical analyses of past software development experiences are crucial. These analyses are applied under the assumption that measures of past software development projects (e.g., defect data) result from similar underlying factors. Thus, it is assumed that data collected from previously developed software can be used to predict the characteristics of future software development projects in similar development environments.

Software defect reports are commonly used to find useful software practices based on past development experience. For the most part, they have been used for either reliability modeling to identify fault-prone software units or prediction of the remaining number of defects, to assess software quality prior to release (Fenton and Neil 1999; Musa et al. 1987; Stringfellow and Andrews 2002). Testing stops when reliability meets a certain requirement (i.e., an acceptably low number of defects) or when the benefit of continued testing does not justify the cost. While this approach is useful, most resulting predictions exclude defect repair time, making it difficult to estimate testing schedules and plan testing resources. This is partly because repair time does not only depend on the number of defects but also the sophistication of the defects and the skill of the fixer. Finally, only a fraction of the available data is used for analysis. Existing work in reliability modeling typically uses mathematical models to exploit quantitative data, primarily defect quantities and complexity metrics of software components. However, defect reports contain other quantitative and qualitative data as well (e.g., locations and types of defects, when they were detected and corrected, and names of the testing and correcting teams). These data should also be taken into account in defect analysis since they may impact understanding of software testing practices that are not currently well understood. Instead of limiting analysis and data usage to existing problem formulation and techniques, our research aims to find a methodology that increases data utilization to elucidate practical software testing principles from past defect reports.

Prediction of defect repair time is a difficult problem and has not been researched as intensively as reliability modeling (Schneidewind 2001). Better understanding of defects will be necessary before we can make predictions about the time required to fix them. Causes of software defects include design flaws, implementation errors, ambiguous requirements, requirements changes, and other factors. 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 use of the software). Furthermore, defects can be found by various groups and in various phases of the software development life cycle. Defects found by the system-testing group may be harder to repair than those found in earlier stages. In some cases errors encountered by users may be difficult to reproduce. Furthermore, predicting defect repair time is especially difficult since repairing one problem may introduce new problems into the system.

This paper presents an empirical approach for predicting defect repair time using defect reports and well-established data mining algorithms. Our main contributions include:

1. formulation of the challenging problem of estimating defect repair time along with an approach for the solution that is potentially useful for testing and for defect management in the final stage of software development,
2. a methodology that extends defect data usage by modeling techniques that incorporate both quantitative and qualitative data, and

3. validation of the proposed approach on a real-world software development project and a comparison with the results of one of the existing analytical modeling approaches.

We present a case study using defect reports obtained from the testing of a large medical software system. The rest of the paper is organized as follows. Section 2 discusses background terminologies and related work. Section 3 describes our case study, the defect data set and performance measures. Section 4 gives details of the proposed approach for modeling defect repair time. Section 5 illustrates our approach using the case study and compares accuracy of results obtained from various predictive models using different data mining algorithms. Section 6 compares the proposed approach with other analytical techniques based on reliability or queuing models. Our conclusions are given in Section 7.

2 Background

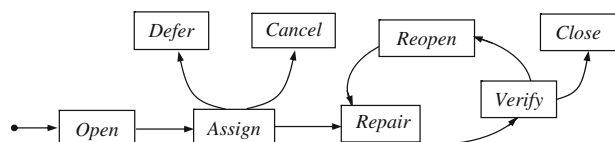
2.1 Software Testing and Defect Life Cycle

Software testing is an important process for system verification and is used to detect the presence of faults but not to guarantee correctness of software (Schach 1996). Various types of testing (e.g., functionality, efficiency, usability) can be applied at multiple levels (e.g., component, integration, system and acceptance) throughout the software life cycle. Testing has various phases (e.g., requirements and design analysis, test planning and development, test execution, and test reporting) and should be performed throughout the software life cycle. However, in this paper we will focus on testing at the implementation level. Acceptance testing typically involves testing a software version by potential customers (*alpha testing*) or by a limited group of users outside the company (*beta testing*). To ensure that repaired software does not cause a regression of previously tested functionality, a *regression test* repeats previously passed tests on the modified software.

Testing allows the discovery of deviations from the system's expected performance, i.e., *software failures*. *Defects* are flaws or errors in the software system that can potentially cause system failure. Failures and defects observed by testing groups or users are documented in *defect reports*, which are reviewed for repair planning. In practice, not all reported defects require repair, since some can be reasonably handled by workarounds in the production environment. Some defects are deferred to future releases and some are simply accepted by customers. In this paper, we use the term defect to refer to any fault or error that causes the software not to function as specified and that requires effort to correct. Software quality assurance should be concerned with defect reduction to assure correctness before other software quality aspects (e.g., reliability, maintainability, portability and usability) are considered.

Defects detected during software testing are recorded in defect reports that keep track of each defect's status until it is corrected. Figure 1 illustrates a life cycle for defects. A defect is born when it is detected and dies when it is corrected.

Fig. 1 A defect life cycle



After a defect is found, it is posted to a defect report. If the lead tester approves, the defect will be *opened* (for repair), otherwise it is either *rejected/canceled* (e.g., if the problem is not an error in the software but is due to the software environment) or *deferred* (e.g., if it is to be corrected in next releases due to a low priority). The lead tester will *assign* the opened defect to a testing team for repair. After being fixed, the defect will be *verified*. If there are no new defects, the defect will be *closed*, otherwise, it will be *reopened* for further repair. The life cycle of defects in our case study has a similar conceptual view but with fewer states.

2.2 Related Work

Empirical research on software defect data analysis has concentrated on the use of reliability modeling to predict the remaining number of defects and/or fault-prone components (Musa et al. 1987; Yamada et al. 1983; Malaiya et al. 1992; Khoshgoftaar et al. 1994; Fenton and Neil 1999; Fenton and Nohlsson 2000; Stringfellow and Andrews 2001, 2002; Li et al. 2003). Traditional approaches typically fit various mathematical models to quantitative data, particularly the number of defects found during different testing periods. Some recent study has extended the use of data in reliability modeling by applying data mining techniques to incorporate qualitative factors into predictions of software defects (Challagulla et al. 2005).

Research on modeling defect repair time has not received as much attention as reliability modeling. This may be due to an unrealistic assumption that, once detected, defects are corrected immediately, which makes defect repair times irrelevant. To achieve greater accuracy in reliability prediction, modeling of defect repair time is needed (Schneidewind 2001). A number of existing modeling approaches rely on queuing theory, where service times correspond to defect repair (or fault correction) times (Musa et al. 1987; Schneidewind 2001; Gokhale and Mullen 2006).

Schneidewind (2001) proposed an integrated framework that incorporates fault correction modeling into reliability modeling and prediction. This extended framework can improve reliability estimates (in terms of not only the remaining number of defects but also the time to fix them), develop stopping rules for testing, and prioritize tests and test resources. The modeling technique relates fault correction to failure prediction by a time delay associated with a defect repair service time. This time delay can be approximated by a random variable that is exponentially distributed (Musa et al. 1987). The framework consists of two basic time-dependent models for estimating the number of failures detected and the number of faults corrected, from which the model for predicting fault correction time can be derived.

Recent work by Mullen (2006) has introduced two measures of repair time: *mean time to repair* and *defect age* (or repair time). His experiments show that a model of software defect age appears to have a lognormal distribution. Further study by Gokhale and Mullen (2006) has applied multi-priority queuing models to estimate defect correction (or resolution) time for each level of severity. In particular, they applied two priority queue models: *preemptive* priority queue and *head-of-the-line non-preemptive* priority queue and introduced a mixed model of the two. Their experiments show that for the most (least) severe defects, mean resolution time is close to that of the preemptive (non-preemptive) priority queue. In general, mean repair time seems to follow the mixed model.

The Schneidewind model and the Gokhale and Mullen model are similar in that they are both quantitative analytical models using the concepts of reliability and queuing theories, respectively. However, they differ in several features and assumptions. Our technique is

more similar to the Gokhale and Mullen model in that our defect repair time model is not developed after the estimated number of faults to be corrected. Furthermore, unlike the above approaches, our technique does not require any modeling assumptions and is not limited to using quantitative data for analysis. Recent research in Hewett et al. (2006) and Hewett and Kulkarni (2006) has applied data mining techniques to the estimation of defect repair times. However, in this paper we employ a different set of machine learning algorithms and provide a comparison to other analytical modeling approaches. Because our data mining models are empirical and not analytical, it would be interesting to examine the results obtained from different concepts. In particular, Section 6 compares our proposed approach with the modeling approaches by Schneidewind, and by Gokhale and Mullen in more detail.

In an attempt to improve the effectiveness of managing testing process, work has been done using mathematical models to estimate “testing effort,” e.g., the widely used method COCOMO II (Boehm et al. 2000). However, measurements of testing effort often consider only the resources required for testing design and execution, and include neither the resources spent on repairing defects (Culbertson et al. 2001) nor the effort of evaluating testing processes. Our work is quite different.

3 Case Study

3.1 Experimental Data

To evaluate our approach, we present a case study based on (Stringfellow and Andrews 2002) with defect reports obtained during the testing of three releases of a large medical record system. The system initially contained 173 software components, and 15 additional components were added over the three releases. Each release included additional components to provide new functionalities. A defect tracking report was recorded when a defect was detected during various testing stages: system testing, integration testing, and maintenance. A single defect could have duplicate records if it is reported multiple times. The report contains both pre-release and post-release defects.

Each data instance represents a defect with attribute values for defect report number, release number, phase in which the defect occurred (e.g., development, test, post-release), test site reporting the defect, the component and the report date, etc. There are a total of 30 attributes, 12 of which were selected by domain experts as relevant to our problem. Table 1 summarizes the attribute descriptions and their corresponding data types, values and sizes. The five date attributes are listed in chronological order. The period between the *EndDate* and the *LastUpdate* includes the testing and repair efforts for defects that escaped testing and were found during customer usage. As shown by Table 1, a defect can be in one of the following states:

- *Open*: the posted defect has been judged legitimate (note that defects caused by environment limitations or ambiguous requirements are not legitimate) and open for repair
- *Working*: the defect is being worked on
- *Verify*: the defect has been repaired and the repair is being tested for correctness; if the defect is no longer present, the repair is verified.
- *Canceled*: the defect is either judged non-legitimate, or has been deferred to a later stage, or repair is omitted due to lack of resources (e.g., time, manpower)

Table 1 Selected defect data characteristics

Attribute	Description	Type	Size and Value
<i>TestingPhase</i>	The testing phase in which the defect was found.	Discrete	8 values: requirement, code inspection, coding, system testing, third party testing, customer testing, acceptance testing, post-release.
<i>Assessment</i>	Assessment by the tester indicating the defect type and/or repair action	Discrete	17 values: program defect, new function, limitation etc.
<i>Component</i>	Component location of the defect	Discrete	75 values
<i>State</i>	Defect management status	Discrete	Six values: open, working, verify, canceled, closed, returned
<i>Severity</i>	Defect severity	Discrete	4 values: 1 (low), 2, 3, 4
<i>OriginID</i>	Person who found the defect	Discrete	70 values
<i>OwnerID</i>	Person assigned to fix the defect	Discrete	57 values
<i>AddDate</i>	Date defect added to the report	Numeric	Between 1995 and 1999
<i>AssignDate</i>	Date assigned to be repaired	Numeric	Between 1995 and 1999
<i>ResponseDate</i>	Date the tester assessed the assigned defect	Numeric	Between 1995 and 1999
<i>EndDate</i>	Date of first repair attempt	Numeric	Between 1995 and 1999
<i>LastUpdate</i>	Date of last update, i.e., completion of repair	Numeric	Between 1995 and 1999

- *Closed*: The repair was successful; the defect is gone
- *Returned*: the defect still exists after efforts to repair it; it re-enters the defect life cycle again

In addition to the above attributes, we derived two new attributes: *Repair-time* and *Num-Defects* representing the time required to fix the defect and the number of defects in each component, respectively. The former is calculated by subtracting *AssignDate* from *LastUpdate*. For our study we use data from the first release, which contains 1,460 defects (data instances or rows).

3.2 Performance Measures

To evaluate the quality of our predictive models, we employ various standard performance measures as suggested in (Menzies et al. 2007); ideally, different measures can provide different insights. In this section, we define the performance measures used in our experiments.

Traditional statistical modeling often uses the *root mean square error (RMSE)* as a measure of how well the proposed model fits the data. RMSE represents the average distance along a vertical line between the data points and the fitted model. In machine learning, modeling involves classification or prediction that associates patterns from data points with classes that express different concepts. One of the most popular performance measures in machine learning prediction is *accuracy (ACC)*, which is defined to be the ratio of the number of correct predictions to the total number of instances in the test sample. However, ACC does not make use of the nature of the incorrect predictions, which can be useful in many domains including software defect analysis. Consider, for example, model that predicts data points into the binary classes: *positive* and *negative*, as shown in Fig. 2.

Fig. 2 Performance metrics in binary classification

		Actual		
		positive	negative	
Predicted	positive	TP	FP	→ Precision
	negative	FN	TN	
		↓ Recall	↓ Specificity	
TP (or hit) FP (or false alarm, Type I error) FN (or miss, Type II error)				
ACC = $(TP+TN) / (TP+FP+TN+FN)$ TPR (recall/sensitivity/hitrate) = $TP / (TP+FN)$ FPR (false alarm rate) = $FP / (FP+TN)$ Specificity = $TN / (FP+TN)$ Precision = $TP / (TP+FP)$				

TP (true positive) represents the number of positive samples correctly predicted, whereas *FP* (false positive) represents the number of negative samples incorrectly predicted as positive. *TN* (true negative) and *FN* (false negative) are defined symmetrically. *ACC* is computed as $(TP+TN)/(TP+FP+TN+FN)$ and does not take into account the difference between *false alarms* (*type I errors*), measured by *FP*, and *missed detections* (*type II errors*), measured by *FN*. The latter value is particularly important in clinical medicine and defect management. Furthermore, *ACC* can be especially misleading when the distribution of the sample class is unbalanced. For example, a model can achieve 90% accuracy on a sample in which only 10% of the data points are positive simply by predicting every instance as negative, even though all of its predictions for positive samples are incorrect.

As shown in Fig. 2, a variety of metrics have been defined to measure different types of errors. In particular, *TPR* (true positive rate) is defined to be the ratio of positives correctly predicted to the actual number of positives, and *FPR* (false positive rate) is defined to be the ratio of negatives incorrectly predicted to the actual number of negatives. *TPR* is also called the *hit rate*, *recall*, or *sensitivity*. *FPR* is also referred to as the *false alarm rate*. Similarly, *specificity* is defined as the ratio of correctly predicted negatives to the actual number of negatives. *TPR* and *specificity* are independent, i.e., neither one tells us anything about the value of the other. Note that the sum of *specificity* and *FPR* is one, and thus, knowing one allows us to calculate the other. Another useful metric is *precision* (*PREC*), the ratio of correctly predicted positives over the total number of samples predicted as positive.

In this paper we also use *AUC* (area under *ROC* curve) as an additional performance metric. The *ROC* curve is a two-dimensional plot between the *TPR* (*Y*-axis) against the *FPR* (*X*-axis) for each cut point in the testing data set. *AUC* has been shown to be a better measure for evaluating predictive ability than accuracy especially when class distribution is not well balanced (Huang and Ling 1995). More researchers have adopted *AUC* as a standard performance metric. When we deal with multi-class classification, all of the above measures are computed by taking an average of measured results over all possible classes.

4 Proposed Approach for Defect Repair Time Modeling

This section proposes a novel approach for predicting defect repair time using previous experience to construct a predictive model. We first formulate the problem introduced in Section 4.1, discuss relevant issues, and then present the analysis approach.

4.1 Problem Formulation

In this study we propose an empirical approach for defect repair time prediction. Consider the selected attributes introduced in Table 1. These attributes are related to different stages of the defect life cycle as shown in Fig. 3.

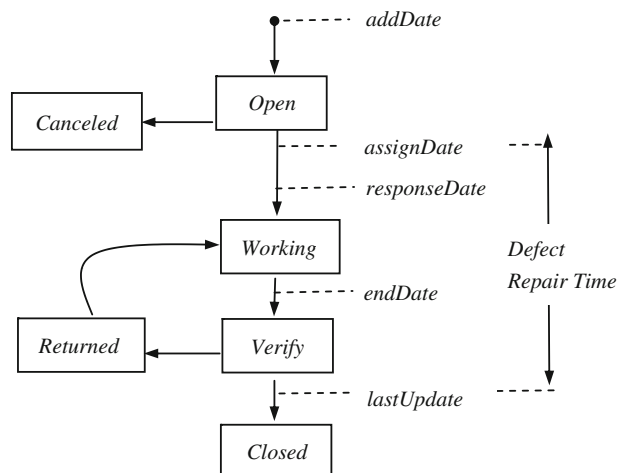
AddDate is the date on which the defect was found and added to the defect report. *AssignDate* represents the date on which a defect was assigned to a programmer. *ResponseDate* is the date on which the programmer, assigned to fix the defect, assessed the defect (e.g., commenting on suitable actions). *EndDate* is the date entered when the first major attempt to fix the defect was completed. Note that *EndDate* may not be the end of the repair process; in most cases the repair must be verified, and if the repair has introduced new problems these must also be handled. *LastUpdate* signifies the end of all work done to fix a particular defect.

Our interest is in predicting *repair-time*, which is defined to be the time period from *assignDate* to *lastUpdate* (as opposed to *endDate*). We ignore the time between *addDate* and *assignDate*, which can be considered as a delay due to management issues, since during this period no one has been assigned to fix the defect. To predict *repair-time* we can construct a predictive model that best fits the data recorded in the defect report. The class (or targeted or labeled) attribute of this data set is *repair-time*, the number of days required to complete repair of a defect. Because the *responseDate* and the *endDate* occur within the period of time that we want to predict, it would be unfair to use them for predicting *repair-time*. Therefore, the data set we use for our predictive modeling consists of all but the last three of the attributes shown in Table 1 along with the class attribute *repair-time*.

4.2 Relevant Analysis Issues

Scarcity of data is a common issue in analyzing aspects of software development. Even when data is available it tends to be incomplete. The problem that we deal with is no exception. There are a number of factors that influence the time required to fix software defects: number of lines of code; number of modules changed or added; complexity of software components, logical controls, or interactions; skill levels of programmers and testers, etc. Unfortunately, many of these factors are not always available in defect reports.

Fig. 3 Timeline for stages relevant to defect management



Furthermore, the data available contains both quantitative and qualitative data types. Many of the relevant factors in defect reports are qualitative (e.g., names of person who fix the defect, component where the defect was found). However, most existing modeling approaches (e.g., statistical and time series techniques) employ quantitative models in which all variables must be continuous or discrete numerical values. A common approach to this problem is conversion of qualitative values into numbers. Although useful in some cases, this imposes an order on the converted values, which may be meaningless or misleading. It is advantageous to have an analysis technique that can directly use qualitative (symbolic) as well as quantitative data.

Data mining research has produced many data analysis techniques that can analyze both quantitative and qualitative data including: association rule mining, Naïve Bayes classification and decision tree learning (Han and Kamber 2000; Kohavi 1995; Quinlan 1993; Witten and Frank 2005). The decision tree is one of the most prominent machine learning techniques that been applied successfully to classification problems with a discrete valued class attribute. Viewing the defect repair problem as a classification problem, we need discrete class attribute values to apply these modeling techniques (e.g., decision trees and the Naïve Bayes technique). In particular, we need to discretize the class attribute, *repair-time*. In this study, we use equal frequency bins (Han and Kamber 2000) to discretize *repair-time*. The resulting values are the three categories: 0–59, 60–103 and more than 103 days. This turns out to reflect the natural clustering in the data set and appears to be meaningful for real software practice. The large grain size of this *repair-time* reflects the fact that in practice a programmer may have to repair multiple defects in the same time period.

4.3 Approaches to Data Analysis

In this study, we employ seven supervised learning algorithms, based on five different approaches, to construct predictive models from a data set. We use several algorithms because we want to know if there exists a feasible solution to the formulated problem, even though a particular algorithm may fail to find it. In practice, we may apply the algorithm that produces the most accurate prediction in a majority of experiments. We select the following approaches because each is based on a different model representation and learning method.

A *Decision tree learner* (Quinlan 1993) is one of the most popular methods in data mining. A *decision tree* is a tree structure whose leaves represent classifications (or predictions) and whose branches represent conjunctions of conditions leading to classifications. To construct a decision tree, the algorithm splits the data source set into subsets based on the value of a splitting attribute (selected by a rank-based measure called *ratio gain*). This process is repeated, in a recursive manner, for each derived subset until either further splitting is not possible, or a singular classification applies to each element of the derived subset. Details of decision table learners are given in (Quinlan 1993; Mitchell 1997).

A *Naïve Bayes Classifier* is based on probabilistic models that incorporate strong independence assumptions that are often violated in reality; hence such classifiers are “naïve”. The model can be derived using Bayes’ theorem. Bayesian classifiers are called active learners, since they require little or no training time. The classification is computed based on the likelihood of each attribute belonging to a particular class. See more details in Mitchell (1997) and Han and Kamber (2000).

K-nearest neighbor is a distanced-based classification algorithm. The algorithm classifies a given query sample by using the the classifications of the *K* training samples

closest to the query sample as “votes” for the class of the query sample. Ties are broken randomly. While this algorithm has the advantage of efficient classification, its drawbacks include selection of K , the appropriate number of neighbors, and the distance metric. A variety of distance metrics has been proposed. See more details in Han and Kamber (2000).

The *Neural net approach* uses a set of connected input/output units (artificial neurons) with weights associated with the connections (Haykin 1995). During a learning phase, the network learns by adjusting the weights to better predict 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 its prediction for each sample with the actual class label. For each training sample, the weights are modified to minimize errors between predicted and actual values. The weight modifications are made in a backward direction, that is, from the output layer to the input layer, hence the name “back propagation”. Neural nets generally take a long time to train and require parameters that are best determined empirically. They do, however, have a high tolerance to noise as well as the ability to adapt to complex (non-linear) patterns.

Support vector machines (SVM) are kernel-based learners that perform classification by finding a hyperplane that optimally separates data into two classes. An optimal hyperplane has the maximum margin between the class boundaries, represented by *support vectors* (Vapnik 1998). To apply SVM for non-linearly separable classes of objects, the objects are mapped into a high dimensional feature space where the two classes become linearly separable. Because the feature space is high dimensional, using feature functions to compute the hyperplane is expensive. Instead, the nonlinear mapping induced by the feature functions is computed with special nonlinear functions called *kernels*. SVMs use a dual representation (i.e., inner products between data objects) and operate in a *kernel* induced feature space. They are closely related to multi-layer perceptron neural networks. Using a kernel function, SVMs can train for polynomial, radial basis function and multi-layer perceptron classifiers by solving a quadratic programming problem with linear constraints, instead of solving a non-convex, unconstrained minimization problem as in neural network training. When dealing with multi-classes, several techniques can be employed, including the “one against many” (classify each class against all other classes) and, the “one against one” (a pair-wise classification giving $n(n-1)/2$ models where n is the number of classes). Although the latter is more computationally expensive, it is more accurate and thus, is the approach employed. See more details of SVMs in Vapnik (1998), and Cristianini and Shawe-Taylor (2000).

5 Experiments and Results

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

5.1 Data Preprocessing

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

For data selection, because we are interested in estimating defect repair time, we select only data points representing 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 limitations. In other words, we consider only defects that are

identified by testers and are completely fixed (i.e., state attribute value is *closed*). After data selection, the data set contains 1,357 data points.

In our data conversion, an attribute with a date value is converted into the number of days between the date and January 1, 1995, which we chose because all the dates in the data start in 1995, and thus the resulting numbers are all positive. Note that using a much earlier date would result in larger positive numbers that would reduce the distinction of different dates and their predictive power on repair time.

Most of our analysis algorithms require a continuous valued class attribute to be discretized. In this study, we apply the *equal frequency bins* technique (Han and Kamber 2000) to discretize *repair-time* into classes, each with approximately the same number of data samples. Having data whose classes are equally distributed helps reduce unbiased training by the learning algorithms.

5.2 Experimentation

We apply seven data mining algorithms: *NaiveBayes*, *3-NN*, *5-NN*, *MultilayerPerceptron*, *SVM*, *DecisionTable* and *J48*, a variation of C4.5 (Quinlan 1993), provided by the data mining tool WEKA (Witten and Frank 2005). These are representative systems for Naïve Bayes classifiers, *k*-nearest neighbors classifiers, neural net learners, kernel-based support vector machines and the decision tree learners, respectively. In addition, we apply *ZeroR*, a majority-voting learner to measure base line performance. The *MultilayerPerceptron* is based on the back-propagation learning algorithm described earlier. For our experiments we used the default setup consisting of a network with one hidden layer (using 34 and 105 neurons for the case *with* and *without* attribute selection, respectively), 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 were obtained using standard empirical procedures. More details on the relevant parameters and their meanings can be found in Witten and Frank (2005).

To avoid overfitting, *n-fold cross-validation*, a standard re-sampling accuracy estimation technique, was used (Kohavi 1995). 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 classifier obtained 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 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 all these *n*-fold cross-validations. For the experiments in this paper, the result is the average result obtained using *n*=10, as suggested in Kohavi (1995).

5.3 Attribute Selection

Although the set of relevant attributes is not extremely large, we apply a standard technique for attribute selection to maximize the accuracy obtained. Our attribute selection technique is based on the *wrapper method* (John et al. 1994). 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 the average accuracy from predictive models obtained from a 5-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 fails to yield a future state with a higher estimated accuracy than that of the current state. It should be noted that, by the nature of hill-climbing search, the set of attributes obtained is not necessary globally optimal.

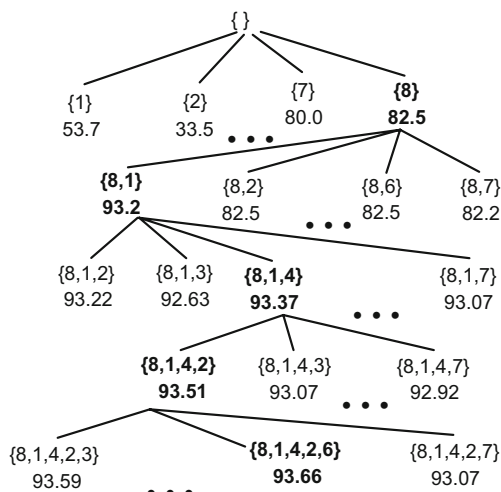
Figure 4 shows a part of the search state space resulting from the wrapper method using the average accuracy obtained from decision tree models as the heuristic evaluation function. We apply the method to a total of eight relevant condition attributes (the top nine attributes of Table 1, according to the data selection, with the omission of the state attribute). Each state represents a set of attributes, and each attribute is represented by its position in Table 1 (e.g., 8 represents the attribute *AssignDate*). The best set at each search level is shown in boldface. The set of selected attributes is {1, 2, 4, 6, 8} representing the attributes *TestingPhase*, *Assessment* (i.e., defect characteristics), *Severity*, *OwnerID* (i.e., programmer), and *AssignDate*, respectively.

5.4 Results

Table 2 shows average percentages, over 10-fold cross validation, of AUC (area under the ROC curve), TPR (true positive rate, recall, sensitivity, hit rate), PREC (precision), FPR (false positive rate, false alarm rate), ACC (accuracy) and RMSE (root mean square) obtained from the eight machine learners. The *ZeroR* learner predicts values based on the majority class and is commonly used as a measure of baseline performance. All results are obtained from the data set with the discretized class attribute values low, medium, and high (i.e., 0–59, 60–103, more than 103 days). Table 2a) and b) show the results obtained using the full set of attributes and the selected set of attributes, respectively. For each performance measure, the best result obtained among various learners is shown in boldface.

Attribute selection appears to improve the results obtained from almost all learners, except for SVM and a decline of less than 1% from NaiveBayes. In particular, results obtained from 5-NN and the neural net improved significantly. As shown in Table 2b), the results obtained from SVM, 3-NN, NaiveBayes are similar while those obtained from 5-NN and the neural net are considerably better. Even though the accuracy obtained from the

Fig. 4 Greedy search for the “best” set of attributes



neural net model increases by close to 28% after attribute selection, it still lags behind the top performer by about 10%. As shown in Table 2, in both cases, the top two performers were the decision table and J48, decision tree models. The decision table model has the highest AUC of 96.63% followed by that of the decision tree model. However, the decision tree model has the highest accuracy, 93.44%, followed by the decision table model with 92.19%. Similar conclusions can be drawn using other performance metrics as well. Both the J48 and decision table models also have high recall and high precision, over 90%, with low false alarm rates, a little over 3%, and root mean square errors of about 20%.

One of the reasons that the different metrics give similar results is the well-balanced class distribution, due to equal frequency binning that discretized *repair-time* into classes containing approximately equal numbers of sample points. It is surprising that SVM performs poorly; it is one of the best performance learners and has increasingly gained popularity in recent years, particularly in Bioinformatics. Perhaps SVM is designed to do well on high dimensional data but not on large sample sets (e.g., experiments with microarray data in Ramaswamy et al. 2001 and Furey et al. 2000). Nevertheless, the overall results obtained from all of the learning models are far better than the baseline performance obtained by random guessing, with accuracy of 33.3%, using ZeroR. Thus, the proposed approach gives promising results, in spite of limited data about the characteristics of software components and defects.

We now do a further comparison of the results obtained from the top two models using confusion matrices and other measures for each *repair-time* intervals (as opposed to averages). In Fig. 5, for each measure, the better result is shown in boldface. Although the overall accuracy of the decision table is slightly lower than that of the decision tree, the decision table predicts long repair times (class value *high*) more accurately than the decision tree. The TPR (recall) results are similar. However, the FPR (false alarm rate) results suggest that the decision tree model performs better for *high* and *medium repair-*

Table 2 Comparisons of average predictions from various learners

Learning approach	AUC	TPR	PREC	FPR	ACC	RMSE
a) Without attribute selection						
NaiveBayes	90.07	77.74	78.49	11.12	77.75	33.91
3-NN	84.47	69.71	70.01	15.14	69.71	38.21
5-NN	84.47	69.71	70.01	15.14	69.71	38.21
Neural net	74.03	55.85	56.44	22.06	55.86	48.66
SVM	87.37	79.59	80.05	10.20	79.59	36.10
J48	95.30	93.15	93.34	3.43	93.15	20.69
Decision Table	96.63	92.19	92.23	3.90	92.19	21.60
ZeroR	49.70	33.33	na	33.33	33.46	47.14
b) With attribute selection						
NaiveBayes	89.70	76.78	77.61	11.60	76.79	33.45
3-NN	86.20	71.33	71.71	14.33	71.33	36.98
5-NN	95.00	79.81	80.07	10.09	79.81	29.10
Neural net	92.33	83.34	83.72	8.32	83.35	30.63
SVM	85.27	77.75	79.04	11.12	77.75	36.47
J48	95.47	93.44	93.64	3.28	93.44	20.43
Decision Table	96.63	92.19	92.23	3.90	92.19	21.60
ZeroR	49.70	33.33	na	33.33	33.46	47.14

na Not applicable

times while the decision table does better for *low repair-time* prediction. Furthermore, the AUC (influenced by TPR and FPR) results suggest that the decision table performs better for all *repair-time* values. In practice, it is possible that the impact of underestimation of lengthy repair times can be more damaging than overestimation of shorter repair times, since project overruns are costly and difficult to manage. In such cases, the decision table model may be a better option than the decision tree in spite of a sacrifice in accuracy and precision. On the other hand, if having a low false alarm rate for *high* and *medium* repair times is essential (e.g., to avoid over-budgeting resources) then the decision tree model may be a better option. Our aim is to illustrate how these results can be interpreted rather than to suggest decisions.

Figure 6 illustrates examples of the rules extracted from the decision tree model constructed using all 1,357 training instances. Each rule represents a path from the root to a leaf. The resulting model is a tree of size 35 with a total of 25 rules. The model has 94.44% accuracy on the training set and has a root mean squared error of 0.204. In Fig. 6, each rule is followed by “: x/y ”, where x and y are the numbers of training instances (with the same condition as the rule condition) correctly and incorrectly predicted by the rule, respectively. Recall that each date is converted into the number of days 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 implies that if the defect was assigned “early,” the repair-time is likely to take longer than 103 days (with support evidences of about 28%). Possible explanations for this are that programmers delay fixing defects when a deadline is not close or that defects found early may be missing functionality that requires more time to implement. Rule 2 suggests that defects not assigned “late” and found by a customer testing group are likely to have long repair-times. Rule 3 says that defects was assigned “late” and found by a system testing group are likely to have short *repair-times*, i.e., no more than 59 days. In this case the defects is not likely to be a major functional defect, and therefore, it is likely to take less time to fix. Similarly, Rule 5 applies to defects found by the developers. These defects are likely to be found at an early stage and so take less time to repair. As shown in Fig. 6, Rule 2 is the weakest of the five rules in terms of degree of support, as evidenced by the number of matching training data instances. Such rules may provide insight into decisions about testing plans or future management of defects. We are surprised not to see rules, with adequate support, in which the programmer

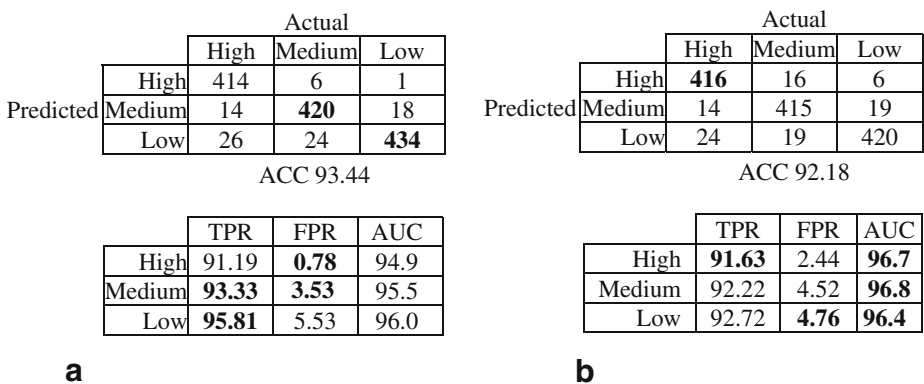


Fig. 5 Comparisons between decision tree and decision table models. **a** Results from J48 decision tree model. **b** Results from decision table model

Fig. 6 Example rules from a predictive model

- Rule 1. assignDate ≤ 909
 $\Rightarrow$ (**repair-time** > **103**): 381/2.
- Rule 2. $909 < \text{assignDate} \leq 951$ and prefix = kt and addDate ≤ 944.5
 $\Rightarrow$ (**repair-time** > **103**): 12/0.
- Rule 3. assignDate > 951 and prefix = bt
 $\Rightarrow$ (**repair-time** ≤ 59): 141/16.
- Rule 4. $909 < \text{assignDate} \leq 998$ and prefix = kt
 $\Rightarrow$ (**59** < **repair-time** $\leq$ **103**): 143/18.
- Rule 5. assignDate > 951 and prefix = bd
 $\Rightarrow$ (**repair-time** ≤ 59): 116/9.

or severity are factors affecting defect repair time. With respect to the effect of programmer identity, this could be because there are 49 programmers in the data, too large a domain value set to form patterns in a set of 1,357 samples. For severity, it may be that severe defects were assigned to better programmers, which would make the degree of severity less revealing and relevant than it should be. Our analysis would probably benefit from information regarding the complexity of each defect repair.

6 Comparisons with Other Analytical Models

6.1 On Model Characteristics and Assumptions

This section compares previous work with our proposed technique in more detail. In particular, we focus on the defect repair time model of two approaches: the S (Schneidewind) model and the GM (Gokhale and Mullen) model. The S model derives repair time by assuming that the number of defects corrected can be modeled the same way as the number of defects detected, and thus, the time required to repair a defect can be computed by the time required to detect it plus a delay. Following the traditional approach, the delay in failure detection is modeled as the delayed *S*-shaped model. In the *S* model, given n , the remaining number of defects to be repaired, the predicted time required for fixing them T_n is given below.

$$T_n = \left(\log \frac{\alpha}{\alpha - \beta(n - n_s - 1)} \right) / \beta + (s - 1 + dT) \quad \text{for } \alpha > \beta(n - n_s - 1) \quad \text{and } n \geq n_s - 1,$$

where s is the starting point of the interval to observe failures, n_s is the observed defect correction count before s (i.e., in $[1, s - 1]$), α is the failure rate at the beginning of interval s , β is the relative rate of change of the failure rate, and dT is assumed to have an exponential distribution.

The GM model applies queuing theory to estimate expected defect resolution time. The defect is considered *resolved* if it is fixed or it is decided that a fix is not needed (e.g., duplicate or irreproducible bugs). This matches our notion of defect repair time. Based on queuing models, defects are analogous to customers whereas the developers who fix the bugs are servers. Thus, repair time includes the time the defect waits (to be repaired) and service (actual repair) time. Defects enter a queuing system consisting of multiple single-server queues of repair service. Although the service is generally FCFS (first come first serve), it is assumed that a defect with high impact on software (high severity) has a higher priority for fixing than those with lower severity. The GM model combines two types of priority queues: *preemptive* and *head-of-the-line non-preemptive* models (Larson and Odoni

2007; Kleinrock 1976). In the former, high-severity defects are never kept waiting in favor of lower-severity ones. Thus, the programmer may stop work on his current defect in order to start work on a newly arrived bug with a higher priority. For the head-of-the-line non-preemptive priority queue, repair service is never interrupted if it has already started. The GM model assumes that each programmer has the same repair service rate, μ . For imperfect repair, the defect will reenter the queue until it has been completely fixed. Thus, the service time of each defect is the period between its *AssignDate* and its *EndDate*. Let t_i and ρ_i represent an expected repair time and a traffic rate (the ratio of defect arrival rate over repair rate) for defects with severity i (with one being the highest), and let ρ be the traffic rate of the overall queuing system. Based on queuing theory concepts (Kleinrock 1976), the expected repair time in the GM model can be expressed as follows (Gokhale and Mullen 2006):

$$\begin{aligned}
 t_1 &= \frac{1}{\mu} \left(1 + \frac{\rho_1}{1 - \rho_1} \right) \\
 t_2 &= \frac{1}{\mu(1 - \rho_1)} \left(1 + \frac{\rho}{\left(1 - \sum_{j=1}^2 \rho_j \right)} \right) \\
 t_3 &= \frac{1}{\mu} \left(\frac{1}{(1 - \rho_1)} + \frac{\rho}{\left(1 - \sum_{j=1}^2 \rho_j \right) \left(1 - \sum_{j=1}^3 \rho_j \right)} \right) \\
 t_4 &= \frac{1}{\mu} \left(\frac{1}{(1 - \rho_1)} + \frac{\rho}{\left(1 - \sum_{j=1}^3 \rho_j \right) \left(1 - \sum_{j=1}^4 \rho_j \right)} \right)
 \end{aligned}$$

Note that the above models assume preemptive priority queue for defects with severity one and non-preemptive priority queue for lower severities. The S and GM models have several features and assumptions that differ from our modeling approach. The basic approaches for the S, GM and our models are based on the delayed S-shape, queuing theory and machine learning models, respectively.

Table 3 summarizes and compares these modeling characteristics. None of the approaches consider repair time for defects that are deferred for later repair if and when they become sufficiently critical. (e.g., in a future release). The S model excludes the possibility that a defect repair may introduce new defects, whereas the other two do not. Unlike the S and the GM models, our model includes repair time of defects detected during customer testing. The S model is driven by the explicit number of defects and thus, can be integrated easily into a reliability assessment framework. On the other hand, it does not include other factors that could influence repair time, such as severity, experience of the programmers or complexity of the defects. The GM and our models do not explicitly model the number of defects but do include other defect repair factors.

Both the S and GM models require modeling assumptions, some of which are not always satisfied in practice. For example, both assume that a defect correction starts immediately after detection. The GM model assumes that all fixers repair bugs at the same rate. These assumptions tend not to hold in general. Our approach does not require these assumptions. Finally, the outputs of the three models are all different. The S model estimates bounds for repair time, whereas the GM predicts expected repair time and ours predicts the repair time interval for each defect based on its characteristics. As shown in Table 3, our proposed approach is more similar to the GM model than the S model, so a comparison may be interesting. Although the GM model and our modeling approaches are

based on different principles and provide different kinds of information from the outcomes, we will attempt to compare them as completely as possible by applying both approaches to a real-world case study.

6.2 Comparisons on Experiments with the Case Study

Before we do the comparison, we first investigate the behavior of our data with respect to Mullen's claim that the software defect age model fits a lognormal distribution (Mullen 2006). Here a defect birth and death corresponds to detection and correction, respectively. Figure 7a shows that the percentages of defects repaired within a given number of days can be approximated closely to a lognormal distribution with mean 4.36 ± 0.001 and standard deviation 0.68 ± 0.002 . These parameters were obtained from the curve fitting utility in the gnuplot version 4.0. In addition, we observe that defect repair time (excluding waiting time), like many queuing service time, exhibits negative exponential random variation as shown in Fig. 7b, where the fitted curve gives an R^2 value of 0.542. (The R^2 value is an indication of how well the model fits the data, e.g., R^2 close to 1.0 indicates that we have accounted for almost all of the variability with the model (Lapin 1973)).

To compare the results obtained from our approach and the GM modeling approach, we will use a train-test sampling method. It should be noted that the GM models presented in Section 6.1 have an implicit assumption that $\sum_{j=1}^k \rho_j < 1$, for each severity k and ρ_j , the

Table 3 Comparisons of three modeling techniques for defect repair time

Modeling characteristics	S model	GM model	Proposed model
Foundations	Delayed S-shaped models	Mixed pre-emptive and non-preemptive priority multiple single server queuing models	Machine learners
Meaning of repair time	Excludes repair time of deferred (to next release) defects	Same as S	Same as S
	Excludes imperfect defect repair	Includes imperfect defect repair time	Same as GM
	Includes only system testing	Includes both system and customer testing	Same as GM
Modeling inputs: factors that influence repair time	Driven by the number of defects to be corrected	Driven by defect severity	Driven by severity and other characteristics (e.g., fixers, testing phase)
Modeling output: estimated repair time	Upper limit—cumulative time to repair a given number of defects to be fixed	Expected repair time for a given defect severity	Repair time interval for each given defect
Modeling assumptions	Defect repair starts when failures are detected	Same as S.	No assumption
	Rate of fault correction is proportional to rate of failure detection	Same as S, i.e., defect repair keeps up with failures detected	No assumption
	No assumption	Repair service rate of each fixer is the same	No assumption

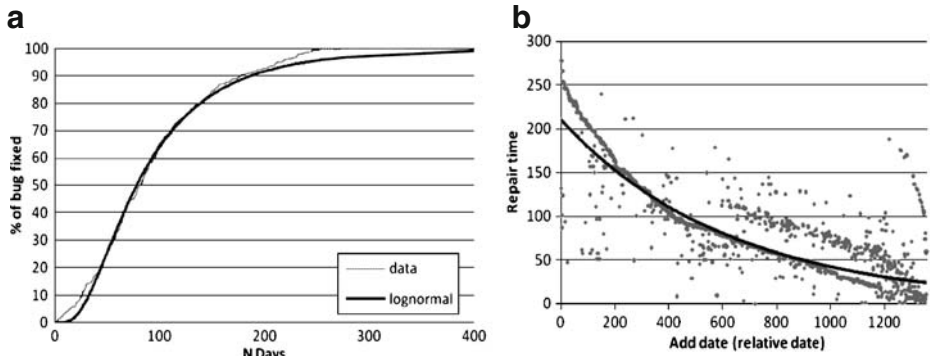


Fig. 7 The repair time on our case study data. **a** Percentages of number of defects repaired per day. **b** Defect repair time (days) corresponding to their AddDate

traffic rate for repair of defects with severity j (otherwise the models predict negative expected repair times). It can be shown that this condition is satisfied if ρ , the traffic rate of the overall queuing system is less than one, i.e., when the defect arrival rate is less than the defect repair (service) rate. In practice, this may not hold at the beginning of the testing phase. However, as testing progresses fewer defects are likely to be found because software becomes more mature and stable, although at the same time defects could also become more sophisticated and take longer to correct. In this experiment, we use day 404 to day 457 as a training period and day 458 to day 507 as a testing period. Although the testing period is longer than the training period, period length itself is not as relevant as the number of defects assigned and fixed during the period. For example, because no defect is assigned beyond the last day of the testing period, extending the testing period further does not change our defect repair time analysis. Our training and testing periods contain 87 and 31 defects, respectively. Thus, a little over 2/3 of all data in both periods is used for training, which is appropriate for the standard train-test sampling approach in machine learning.

From the training data set, we compute the expected repair time for each severity using the GM model given in Section 6.1. Note that in the GM model, when a defect is not properly fixed, it re-enters as a new arrival into the queue. Thus, the repair time is measured from the *assignDate* to the *endDate* (as opposed to the *lastUpdate*), and therefore the number of defects fixed in a given time period is equivalent to the number of defects whose *endDate* falls within the period.

We first need to compute ρ , the traffic rate of the overall queuing system, which is the ratio between λ , the *arrival rate*, and τ , the *service rate*, of the overall queuing system. We can estimate λ by the ratio of the number of defects assigned during the training period to the number of days in the training period, l . Similarly, τ can be estimated by the ratio of the number of defects fixed during the training period to l . For the case study data, we obtain λ and τ values of 1.66/day and 2.26/day, respectively. This gives the estimated value of ρ as 0.74. There were 23 programmers assigned to fix the defects during the training period. We assume the same number of programmers complete all defect corrections in this period. Since the GM models assume that each programmer has the same service rate μ , μ can be estimated from $\tau/23$, which is 0.098/day. For each severity i , we want to estimate the traffic rate ρ_i of each queue (programmer) and apply the GM model to estimate the expected repair time t_i . Here $\rho_i = \lambda_i / \mu$, where λ_i is the arrival rate of each queue for defects of severity i , which can be estimated by $p_i \lambda / 23$, where p_i represents the likelihood that a defect has severity i . Thus, p_i can be estimated as the ratio of the number of defects of severity i to the

Table 4 Results obtained from the GM models

Severity, i	No. of defects	Severity likelihood, p_i	Arrival rate, λ_i	Traffic rate, ρ_i	Expected repair time, t_i
1	15	0.17	0.012	0.127	84.30
2	33	0.38	0.027	0.280	39.72
3	31	0.36	0.026	0.263	22.97
4	8	0.09	0.007	0.068	10.93

total number of defects. Table 4 shows the results obtained. Note that severity 4 represents the highest severity in our case study data and is equivalent to severity 1 in the GM models.

As shown in Table 4, severity 4 has highest priority and thus, the prediction of its expected repair time is the least time, whereas severity 1 takes about eight times as long. Next we compare the GM predictions to actual repair times and the predictions from our proposed approach. Because of the nature of the outputs from the GM models, which are expected repair times by severity, we select to use neural net modeling to produces numeric outputs for comparison. Applying other classification machine learners that predict intervals for repair times will not give a fair comparison.

Table 5 compares actual average repair time for each severity with results predicted by the GM and neural net models. Here the RMSE is the root mean square error between severity groups. (Using the RMSE based on each defect data point will not give a fair comparison to the GM approach.) As shown in Table 5, both approaches perform better in the training period than in the testing period, as expected. In the training period, the GM models tend to overestimate repair times for low severity defects and underestimate the correction time for high severity defects, whereas the neural net approach overestimates in all severity levels but with much smaller error gaps. In the testing period, the GM models overestimate repair times except for defects of severity 3, whereas the neural net approach underestimates repair times of lower severity groups and overestimates for the high severity group. The RMSEs obtained from the neural net models are significantly lower (about 50% lower) than those of the GM model in both training and testing periods.

To compare the percentages of the RMSEs obtained in Table 2, we normalize them to have a range between zero and one, as do the RMSEs in the classification results. As shown in Table 5 b), the RMSE obtained from the neural model is 27.65, whereas the best RMSE

Table 5 Comparisons of predicted results

Severity, i	Actual average repair time	GM's expected repair time	NN's average repair time
a) Training error			
1	61.96	84.30	69.27
2	27.90	39.72	34.10
3	28.69	22.97	36.56
4	16.48	10.93	22.09
RMSE		13.25	6.8
%Normalized RMSE		31.17	16.01
b) Testing error			
1	41.79	84.30	27.73
2	16.36	39.72	13.91
3	36.82	22.97	18.30
4	5.86	10.93	8.18
RMSE		25.35	11.75
%Normalized RMSE		59.63	27.65

obtained from the classification by neural net in the general case, as shown in Table 2b), is 30.63. The small difference, about 3%, indicates that our earlier estimates give reasonably representative results for generalizing the data behavior.

The GM model performs extremely well with the customer defect data, as reported in Gokhale and Mullen (2006), but does not appear to perform well in our case study data. This may be due to the fact that GM models are sensitive to the stability and maturity of the software development environment. Immature software can create outliers in defect data that upset the estimated service time and waiting time of the priority queue in the GM modeling approach. On the other hand, the neural net approach is designed to adapt its modeling capability to learn from the training data employed.

7 Conclusion

We present a novel approach to modeling software defect repair time by applying data mining to software defect data. This approach has potential benefits for defect management, including test resource allocation and scheduling. The model obtained can be used for prediction of defect repair times, which can be used to help avoid overruns in software projects (due to overly optimistic schedules). We illustrate our approach with a case study. The results of our predictive models, obtained from various data mining algorithms, are promising. The averages of the best AUC and ACC values are 96.63% and 93.44%, respectively. However, as with any modeling approach based on historical data, our approach has the inherent limitation that the resulting models can only be applied to software projects with the same characteristics (i.e., they must have the same schema, the set of condition attributes, and attribute domains). One remedy for this is to build predictive models using several defect data sets collected from different types of projects (e.g., organic, embedded as in COCOMO II (Boehm et al. 2000)). Extending the generality of the model's application in this way is an object of our future research.

Acknowledgements The authors would like to thank the referees and John Leuchner for their helpful comments, which have improved the quality of this paper. Special thanks to Anneliese Andrews and Catherine Stringfellow for providing the case study data used to illustrate the approach.

References

- Boehm B, Basili V (2001) Software defect reduction top 10 list, *Software Management*. pp 135–137
- Boehm B, Horowitz E, Madachy R, Reifer D, Clark B, Steece B, Brown A, Chulani S, Abts C (2000) *Software cost estimation with COCOMO II*. Englewood Cliffs, Prentice-Hall
- Challagulla V, Bastani F, Yen I, Paul R (2005) Empirical assessment of machine learning based software defect prediction techniques. In *Proceedings of the 10th IEEE International Workshop on Object-oriented Real-time Dependable Systems*
- Cristianini N, Shawe-Taylor J (2000) *An introduction to support vector machines*. Cambridge University Press, Cambridge, UK
- Culbertson R, Brown C, Cobb G (2001) *Rapid Testing*. Prentice Hall, Upper Saddle River, NJ
- Fenton N, Neil M (1999) A critique of software defect prediction models. *IEEE Trans Softw Eng* 25(5):675–689
- Fenton N, Ohlsson N (2000) Quantitative analysis of faults and failures in a complex software system. *IEEE Trans Softw Eng* 26(8):797–814

- Furey T, Christianini N, Duffy N, Bednarski D, Schummer M, Haussler D (2000) Support vector machine classification and validation of cancer tissue samples using microarray expression data. *Bioinformatics* 16(10):906–914
- Galen R (2005) *Software endgames: eliminating defects, controlling change, and the countdown to on-time delivery*. Dorset House Publishing, NY
- Gokhale S, Mullen R (2006) Queuing models for field defect resolution process. In *Proceedings of the 17th IEEE International Symposium on Software Reliability Engineering (ISSRE-06)*, Raleigh, NC
- Han J, Kamber M (2000) *Data mining: concepts and techniques*. Morgan Kaufmann, San Mateo, CA
- Haykin S (1995) *Neural networks: a Comprehensive Foundation*, 2nd edn. Springer, New York
- Hewett R, Kulkarni A (2006) Alternative approach to utilize software defect reports. In *Proceedings of the 15th International Conference on Software Engineering and Data Engineering (SEDE-2006)*. Los Angeles, CA
- Hewett R, Kulkarni A, Stringfellow C, Andrews A (2006) Software defect data and predictability for testing schedules. In *Proceedings of the 18th International Conference on Software Engineering and Knowledge Engineering*, San Francisco, CA
- Huang J, Ling CX (1995) Using AUC and accuracy in evaluating learning algorithms. *IEEE Trans Knowl Data Eng* 17(3):299–310 2005
- John G, Kohavi R, Pfleger K (1994) Irrelevant features and the subset selection problem. In *Proc. of the 11th International Conference on Machine Learning*, pp 121–129
- Khoshgoftaar T, Szabo R, Woodcock T (1994) An empirical study of program quality during testing and maintenance. *Softw Qual J* 3:137–151
- Kleinrock L (1976) *Queueing systems, Volume II: computer applications*. Wiley, New York
- Kohavi R (1995) The power of decision tables. In *Proceedings of European Conference on Machine Learning*. Springer-Verlag
- Lapin L (1973) *Statistics for modern business decisions*. Harcourt Brace Jovanovich, Orlando, FL
- Larson R, Odoni A (2007) *Urban operations research* (Prentice-Hall, NJ, 1981). Dynamic Ideas, Belmont, MA
- Li P, Shaw M, Herbsleb J (2003) Selecting a defect prediction model for maintenance resource planning and software insurance. *Proceedings of the Fifth Workshop on Economics-Driven Software Research*, IEEE Computer Society, pp 32–37
- Malaiya Y, Karunanithi N, Verma P (1992) Predictability of software reliability models. *IEEE Trans Reliab* 41(4):539–546
- Menzies T, Dekhtyar A, Distefano J, Greenwald J (2007) Problems with precision: a response to “Comments on ‘Data Mining Static Code Attributes to Learn Defect Predictors’”. *IEEE Trans Softw Eng* 33(9):637–640
- Mitchell T (1997) *Machine learning*. McGraw-Hill, NY
- Mullen R (2006) Characterizing software defect repair time. In *Proceedings of the 17th IEEE International Symposium on Software Reliability Engineering (ISSRE-06)*, Raleigh, NC
- Musa J, Iannino A, Okumoto K (1987) *Software reliability: measurement, prediction, application*. McGraw-Hill, NY
- Quinlan R (1993) *C4.5: programs for machine learning*. Morgan Kaufmann, San Mateo, CA
- Ramaswamy S, Tamayo P, Rifkin R, Mukherjee S, Yeang C, Angelo M, Ladd C, Reich M, Latulippe E, Mesirov J, Poggio T, Gerald W, Loda M, Lander E, Golub T (2001) Multiclass cancer diagnosis using tumor gene expression signatures. *Proc Natl Acad Sci U S A* 98:15149–15154
- Schach S (1996) *Testing: principles and practices*. ACM Comput Surv 28(1):277–279
- Schneidewind N (2001) Modeling the fault correction process. In *Proceedings of International Symposium on Software Reliability Engineering (ISSRE 2001)*, pp 185–190
- Stringfellow C, Andrews A (2001) Quantitative analysis of development defects to guide testing. *Softw Qual J* 9(3):195–214
- Stringfellow C, Andrews A (2002) An empirical method for selecting software reliability growth models. *Empirical Software Engineering* 7(4):319–343
- Vapnik V (1998) *Statistical learning theory*. Wiley-Interscience, New York
- Witten I, Frank E (2005) *Data mining practical machine learning tools and techniques*. Morgan Kaufmann, San Francisco, CA
- Yamada S, Ohba M, Osaki S (1983) S-shaped reliability growth modeling for software error decision. *IEEE Trans Reliab* 32:475–478



Rattikorn Hewett is a professor in the department of Computer Science, Texas Tech University. She has a Ph.d. in Computer Science from Iowa State University, an M. Eng. Sc. in Computer Science from the University of New South Wales, and a B.A. (Hons) in Pure Mathematics and Statistics from Flinders University, Australia. She was a postdoctoral fellow at the Knowledge System Laboratory, Stanford University. Her research in Artificial Intelligence (AI) includes model-based reasoning, intelligent control, blackboard architectures and machine learning with applications in data mining, bioinformatics and software engineering. Her recent work involves applied AI research to automated software engineering in the areas of software quality assurance, software testing, and security analysis and risk assessment of software architectures. She has published extensively and has served on numerous international conference and workshop program committees.



Phongphun Kijsanayothin is a Ph.D. candidate in the department of Computer Science, Texas Tech University. He has an M. Eng. (Computer) from Kasetsart University and a B. Eng. (Computer) from the King Mongkut's Institute of Technology Ladkrabang, Thailand. His research areas include software security informatics and software testing in the context of object-oriented development. He is a member of IEEE.