Evolutionary Optimization of Software Quality Modeling with Multiple Repositories

Yi (Cathy) Liu, *Member*, *IEEE Computer Soceity*,
Taghi M. Khoshgoftaar, *Member*, *IEEE*, and Naeem Seliya, *Member*, *IEEE*

Abstract—A novel search-based approach to software quality modeling with multiple software project repositories is presented. Training a software quality model with only one software measurement and defect data set may not effectively encapsulate quality trends of the development organization. The inclusion of additional software projects during the training process can provide a cross-project perspective on software quality modeling and prediction. The genetic-programming-based approach includes three strategies for modeling with multiple software projects: Baseline Classifier, Validation Classifier, and Validation-and-Voting Classifier. The latter is shown to provide better generalization and more robust software quality models. This is based on a case study of software metrics and defect data from seven real-world systems. A second case study considers 17 different (nonevolutionary) machine learners for modeling with multiple software data sets. Both case studies use a similar majority-voting approach for predicting fault-proneness class of program modules. It is shown that the total cost of misclassification of the search-based software quality models is consistently lower than those of the non-search-based models. This study provides clear guidance to practitioners interested in exploiting their organization's software measurement data repositories for improved software quality modeling.

Index Terms—Genetic programming, optimization, software quality, defects, machine learning, software measurement.

1 Introduction

Software quality improvement methods have a valuable role in software engineering practice [1]. Some of these methods include code inspections, design walkthroughs, prototype simulation, and measurement-based analysis. Practitioners are often interested in identifying problematic areas in their software, with the goal of maximizing benefits from the limited software quality improvement resources. Software quality modeling generally involves predicting low-quality areas of the software product, thereby assisting the design and testing team in focusing their quality improvement tasks.

Software quality models generally predict, for a program module, either the number of defects it is likely to have or the quality-based risk category it belongs to, e.g., fault-prone (*fp*) or not-fault-prone (*nfp*) [2], [3], [4]. In the literature, various classification techniques have been applied for software quality modeling, such as Logistic Regression [5], Naive Bayes [6], and Decision Trees [7], [8]. Software measurement and defect data from a prior software release or similar project are used to train the software quality model, which is then applied to predict the quality of the target system with known software metrics.

In the software industry, it is common for an organization to maintain several software metrics repositories for projects developed [9]. The data in these repositories are likely to follow similar patterns, especially if the organization enforces the same development life cycle, as well as the same coding and testing practices. While most existing related works focus on training using one software measurement data set, we emphasize including all relevant past projects during the training process. The working hypothesis is that multiple software repositories provide additional information that can improve the predictive performance of the trained software quality model.

A common problem during software quality modeling is searching for an optimum model that adequately satisfies quality improvement goals. For example, the different costs of misclassifying *fp* and *nfp* modules poses model selection challenges. The search for an optimal solution is compounded when modeling with multiple software project data sets. We present two case studies of building software quality models with multiple software data repositories. The multiple software project data are obtained from the NASA software metrics data program, and include seven software measurement data sets with known defect data.

The first case study involves Genetic Programming (*GP*) as a search-based software engineering technique [10], [11] for determining a practical and robust software quality model. *GP* follows the Darwinian principle of survival and reproduction of the fittest individuals [12] and performs a parallel searching process for each given computation problem. A common problem associated with *GP*-based machine learning is overfitting, where generalization capability is relatively poor compared to prediction on the

- *Y. Liu is with The J. Whitney Bunting School of Business, Georgia College and State University, 231 W. Hancock St, Milledgeville, GA 31061. E-mail: yi.liu@gcsu.edu.*
- *T. M. Khoshgoftaar is with the Department of Computer and Electrical Engineering and Computer Science, Florida Atlantic University, 777 Glades Road, Boca Raton, FL 33431. E-mail: taghi@cse.fau.edu.*
- *N. Seliya is with the Computer and Information Science Department, University of Michigan–Dearborn, 4901 Evergreen Rd, Dearborn, MI 48128. E-mail: nseliya@umich.edu.*

Manuscript received 1 Sept. 2008; revised 2 Sept. 2009; accepted 21 Sept. 2009; published online 12 May 2010.
Recommended for acceptance by M. Harman and A. Mansouri.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number TSESI-2008-09-0273.
Digital Object Identifier no. 10.1109/TSE.2010.51.

training data [13]. A validation data set added during training is often used to counter model overfitting [14], [15]. In contrast, we investigate a multiple data sets validation process during training. We build *GP*-based software quality models using three strategies for dealing with multiple software measurement data sets—as discussed in Section 4.

The second case study involves 17 non-*GP* classification techniques applied to (the same) multiple software metrics data sets [16]. Learners from different computational theories are included, such as probabilistic learning, rule-based learning, ensemble-query learning, and simple prediction models. Each learner is applied to the seven software measurement data sets and a voting scheme is used to predict the quality-based class of program modules in the test data set. A discussion is presented for comparing both case studies for their optimal solutions to software quality modeling with multiple software project data repositories. It is shown that the search-based software quality modeling provides more benefit to the software practitioner in terms of both expected cost of misclassification and software quality prediction.

The remainder of the paper is structured as follows: Section 2 summarizes key points of software quality classification, including expected costs of misclassification and model selection strategy. Section 3 presents the seven software measurement data sets of our case studies. Section 4 details the *GP*-based solutions for software quality modeling with multiple software repositories. Section 5 summarizes the methodology followed in our second case study. Section 6 presents the various results of our case studies, including a comparative discussion and threats to empirical validity. Finally, in Section 7, we conclude our work and provide suggestions for future work.

2 Software Quality Modeling

2.1 Software Metrics

Software metrics are proven tools for aiding in project management and process improvement [9], [17], [1], [18]. Prior to project planning, various objectives and goals need to be established for both project management and software quality assurance. In the case of the latter, software metrics assist in the understanding of the technical process of software development. By measuring various characteristics of the software product and its development process, actions can be taken to increase software quality and reliability. While the exact science of software measurements still tends to be controversial, such as the absence of a universal set of best metrics, it is widely accepted that we need to measure internal attributes of software to manage and change external characteristics of software.

Software attributes can characterize the software quality of both the product and the process of software development. Attributes of software quality, such as defect density and failure rate, are external measures of the software product and its development process. We focus on utilizing software metrics, such as code-level measurements and defect data, to build defect predictors or software quality models. This is based on the practical assumption that these software metrics will capture the quality of the end product, more specifically, how the defects are distributed among the program modules of a given system and what makes certain modules more likely to have defects than others. By tapping into historical project data, characterized in the form of software metrics and defect data, we build effective software quality prediction models.

2.2 Software Quality Classification

A low-quality or *fp* prediction can justify the application of available quality improvement resources to those programs. In contrast, an *nfp* prediction can justify nonapplication of the limited resources to these already high-quality programs. The practical goal is to achieve high software reliability and quality with an effective use of available resources. In a two-group classification model such as *fp* and *nfp*, a software quality model can have four prediction outcomes for a target program module: true positive, i.e., correct classification as *fp*, true negative, i.e., correct classification as *nfp*, false positive, i.e., incorrect classification as *fp*, and false negative, i.e., incorrect classification as *nfp*. In terms of misclassification error types, a false positive classification is a Type_I error, while a false negative classification is a Type_II error.

The costs of these two misclassifications are clearly different, as they both have different implications. A Type_II error implies a missed opportunity to detect and rectify a low-quality program module, whereas a Type_I error implies wasted resources since a high-quality module is subjected to unnecessary quality improvement tasks. Hence, for a given software quality model, its (total) expected cost of misclassification (*ECM*) provides practical guidance to the practitioner, where a lower *ECM* value is preferred.

The expected cost of misclassification of a model is defined as:

$$ECM = C_I N_I + C_{II} N_{II}, \tag{1}$$

where C_I is the cost of a Type_I misclassification, C_{II} is the cost of a Type_II misclassification, N_I is the number of Type_I errors, and N_{II} is the number of Type_II errors. Generally, the magnitude of C_{II} can be anywhere from five to 100 times that of C_I, depending on the type of software system, e.g., for high-assurance systems, C_{II} can be 15-50 times C_I [8]. For simplicity, we compute the normalized expected cost of misclassification as $NECM = \frac{ECM}{N}$, where N is the number of program modules in the test data set.

In addition to a low *ECM* value, the practitioner is interested in the Type_I and Type_II error rates because they are associated with the predicted numbers of *fp* and *nfp* modules. For a given classification technique, the Type_I and Type_II error rates are inversely proportional [8]. Thus, a strategy for model selection is needed during model training and evaluation. Consistent with our prior related works, our preferred model selection strategy is to obtain relatively similar Type_I and Type_II error rates with the latter being as low as possible [19]. This strategy is subject to change based on the software quality improvement objectives of the project.

In the literature, one can find various metrics by which performance of a classification model can be gauged. In

addition to the expected cost of misclassification and the two error rates which we use in our study, another performance metric recently used in software engineering literature is the area-under-ROC-curve (*AUC*), i.e., the area under receiver-operating-characteristic (ROC) curve [6], [20], [21], [22]. While *AUC* provides a good singular measure to compare the overall performance of competing models, it does not provide the practitioner with an intuitive meaning. The analyst is more interested in the error rates and predicted number of *fp* modules at a given cost ratio ($\frac{C_{II}}{C_I}$) that is reflective of their software project. This provides practical value to the software quality assurance team. By providing a comparison based on error rates and expected cost of misclassification, we arm the analyst with practical software quality information.

3 Software Measurement Data

The software metrics and quality data used in our study originate from seven NASA software projects. Even though these projects are not directly dependent on each other, they share commonalities other than originating from the same organization. More specifically, they are all targeted to high assurance and complex real-time systems. Therefore, it is practical and relevant to leverage the information spread across these data sets in order to predict the quality of an ongoing similar project.

The data sets were obtained through the NASA Metrics Data Program, and include software measurement data and associated error data collected at the function level [23]. These and other software project data sets are publicly available under the PROMISE software engineering repository [24]. The types and numbers of software metrics made available are determined by the NASA Metrics Data Program. Each instance of these data sets is a program module. The quality of a module is described by its *Error Rate*, i.e., number of defects in the module, and *Defect*, whether or not the module has any defects. The latter is used as the class label.

We only selected 13 primitive software metrics for our study: three McCabe metrics (Cyclomatic_Complexity, Essential_Complexity, and Design_Complexity), five metrics of Line Count (Loc_Code_And_Comment, Loc_Total, Loc_Comment, Loc_Blank, and Loc_Executable), four basic Halstead metrics (Unique_Operators, Unique_Operands, Total_Operators, and Total_Operands), and one metric for Branch Count. Other derived and nonprimitive metrics, such as Halstead's Effort and Halstead's Volume, are not considered during modeling as it was felt that they would not impart any additional knowledge to what is captured by the software quality model from the four primitive Halstead metrics.

Classifiers are built using the 13 software metrics as independent variables and the module class as the dependent variable (i.e., *fp* or *nfp*). In addition to the 13 product metrics, two additional independent variables (or software attributes) were added to the respective data sets. The first attribute indicated the size of the (one of seven) software project that a program module belonged to. We categorized the seven data sets into small, medium, and large sizes based on the number of modules in each data set. The second variable is a Boolean metric representing whether or not an instance belonged to a data set of an object-oriented software system.

TABLE 1
Summary of the Software Data Sets

Dataset	*nfp*	*fp*	Total	Language
KC1	1782	325	2107	C++
KC2	414	106	520	C++
KC3	415	43	458	Java
CM1	457	48	505	C
MW1	372	31	403	C
PC1	1031	76	1107	C
JM1	7163	1687	8850	C

It is important to note that the software measurements are primarily governed by their availability, the internal workings of the respective projects, and the data collection tools used by the projects. We only use functionally oriented metrics for all software data sets, solely because of their availability. This is an unfortunate case of a real-world software engineering situation where one has to work with what is available rather than the most ideal situation. The use of specific software metrics in the case study does not advocate their effectiveness—different software projects may collect and consider different sets of software measurements for analysis [2], [9].

We note that the selection of a best set of predictors in estimation problems has been an ongoing subject of study in software engineering. For example, Cuadrado-Gallego et al. [25] consider an approach to improve the selection of cost drivers in parametric models for software cost estimation. They analyze various factors that affect the importance of a cost driver, and use empirical evidence to formulate an aggregation mechanism for cost driver selection. In the context of software quality classification, Menzies et al. [6] summarize that, instead of selecting a best set of software quality indicators, empirical studies should focus on building software quality classification models that are useful and practical. They summarize that the best attributes to use for defect prediction vary from data set to data set and are project-specific, confirming a relatively similar observation made by others [26], [27].

The data sets are related to projects of various sizes written with various programming languages. Table 1 summarizes the seven data sets used in this case study. Those data sets are referred to as JM1, KC1, KC2, KC3, CM1, MW1, and PC1. It is worth mentioning that the KC data sets are written using object-oriented languages. Since this study focuses only on a unirepresentation approach (i.e., same features across data sets), object-oriented metrics provided by the NASA Metric Data Program are not considered. Each software system and its data set is briefly described below:

- **KC1** is a project that is comprised of logical groups of computer software components (CSCs) within a large ground system. KC1 is made up of 43 KLOC in C++. The data set contains 2,107 instances, and of these instances, 325 have one or more faults and 1,782 have zero (i.e., unreported) faults. The maximum number of faults in a module is seven.

- **KC2** is a C++ program, with metrics collected at the function level. The KC2 project is the science data processing unit of a storage management system used for receiving and processing ground data for missions. The data set includes only those modules that were developed by NASA software developers. The data set contains 520 instances, and of these instances, 106 have one or more faults and 414 have zero faults. The maximum number of faults in a software module is 13.
- **KC3** has been coded in Java and has 18 KLOC. This software application collects, processes, and delivers satellite metadata. The data set contains 458 instances, and of these instances, 43 have one or more faults and 415 have zero faults. The maximum number of faults in a module is six.
- **CM1** is written in C with approximately 20 KLOC. The data available for this project is from a science instrument. It contains 505 instances, and of these instances, 48 have one or more faults and 457 have zero faults. The maximum number of faults in a module is five.
- **MW1** is the software from a zero gravity experiment related to combustion. The experiment is now completed. It is comprised of 8,000 lines of C code. The data set contains 403 modules, and of these instances, 31 have one or more faults and 372 have zero faults. The maximum number of faults in a module is four.
- **PC1** is flight software from an earth orbiting satellite that is no longer operational. It consists of 40 KLOC in C. The data set contains 1,107 instances, and of these instances, 76 have one or more faults and 1,031 have zero faults. The maximum number of faults in a module is nine.
- **JM1** is a real-time C project which has approximately 315 KLOC [9]. There are eight years of error data associated with the metrics. The changes to the modules are based on the changes reported within the problem reports. We processed JM1 to eliminate redundancy, obvious noisy instances, and observations with missing values. The preprocessed data set contains 8,850 modules, and of these instances, 1,687 have one or more faults. The maximum number of faults in a software module is 26.

4 Case Study 1 Methodology

A typical *GP* process initiates with a number of individuals (each is a potential solution) generated randomly. A fitness function is used to assess the solution optimality of a given individual, i.e., to determine how well an individual is able to solve the given problem. Mimicking natural evolution, a *GP* process runs for a given number of generations and uses various genetic operators to propagate from one generation to the next. Individuals with better fitness are more likely to survive and pass their characteristics on to the next generation. When the *GP* process is completed, the best individual generated will be the solution of the given problem. Since a given *GP* process may not yield the optimal solution, multiple *GP* processes are usually required. However, one should note that even multiple runs of GP cannot always guarantee the optimality of the solution search process.

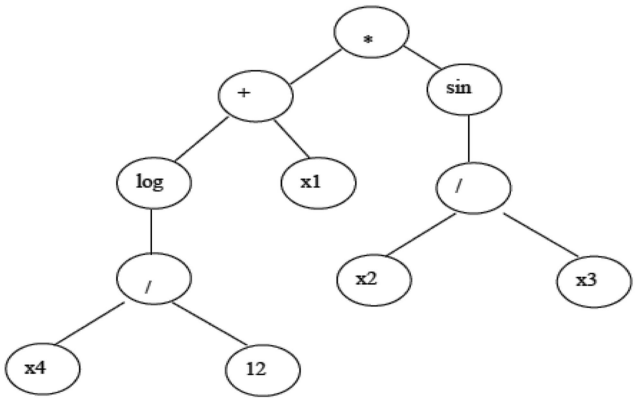

Fig. 1. Example GP solution.

4.1 Genetic Programming Process

Each individual is an S-expression-tree composed of functions and terminals provided by the analyst for a given problem [28]. Examples of functions include sine, cosine, add, and subtract, while examples of terminals include true, false, X, and Y. An example of a GP solution from our case study is shown in Fig. 1, which represents the expression $(\log(\frac{x_4}{12}) + x_1) * (\sin(\frac{x_2}{x_3}))$. In the context of our case studies, x_1 is the $\mathrm{Cyclomatic\_Complexity}$, x_2 is the $\mathrm{Essential\_Complexity}$, x_3 is the $\mathrm{Design\_Complexity}$, and x_4 is the $\mathrm{LOC\_Code\_Comment}$. The stepwise description of our *GP* process is presented below:

- Initialization: The first step of a *GP* process is initialization, where the first population in the problem domain is generated randomly. Generally, individuals in the first population have extremely poor fitness. The first population can be generated using the half-and-half method [29], i.e., 50 percent of individuals in the first population are generated using the full method, where all individuals have the specified maximum depth of trees. The other 50 percent of individuals are produced using the grow method, where the individuals can have various shapes, but the length of each branch from the endpoint to the root in every individual cannot be greater than the specified maximum depth.
- Evaluation: Each individual is evaluated and assigned a fitness value based on the fitness function [12]. The two measures of fitness used in our study are the raw fitness and adjusted fitness. The raw fitness uses the natural terminology of a problem to express the fitness. It generally measures the amount of errors in an individual's attempted solution, e.g., the number of modules that are misclassified or the total cost of misclassification. The adjusted fitness $a(i,t)$ is computed as

$$a(i,t) = \frac{1}{1 + s(i,t)}, \tag{2}$$

where, for the tth individual in the ith generation, $s(i,t)$ is the standardized fitness which is computed by transforming the raw fitness such that a lower numerical value implies a better fitness of the individual. The range of the fitness function is $(0,1]$, where 1 indicates the highest fitness level. The reason for using the adjusted fitness is that it can exaggerate a small difference when the standardized fitness approaches zero.

- Selection: Once the performance of each individual is known, a selection algorithm is carried out to select individuals. Three commonly applied selection algorithms are:
  - Fitness-Proportional Selection: With such a selection, the possibility of an individual being selected depends on his ability compared to others in the population. It is calculated as

$$p_i = \frac{f_a(i)}{\sum_{k=1}^{n} f_a(k)},$$

    where $f_a(i)$ is the adjusted fitness of an individual. The higher the adjusted fitness of an individual, the greater the probability of it being selected. Every individual has some opportunity to be selected.
  - Greedy Overselection: The method may help GP achieve better performance when the size of a population is large. All of the individuals in the population are sorted, in decreasing order, based on their normalized fitness. The individuals are then divided into two groups. The individuals in the first group are the fitter individuals, accounting for a certain percent of the normalized fitness. The less fit individuals are placed into the second group. In 80 percent of runtimes, an individual is selected from the first group. The selection method inside the two groups is Fitness-Proportional Selection. The method may speed up the convergence of a GP run and reduce the number of generations needed for successfully finding a solution.
  - Tournament Selection: Instead of exhaustively testing every member of the population, a binary tournament is run to determine a relative fitness ranking. Two members are chosen at random from the initial population to compete with each other, and the winner proceeds to the next level of the tournament. When the tournament is over (i.e., only one individual is left), the relative fitness of each member of the population is awarded according to the level of the tournament it has reached. This method allows us to adjust the selection pressure by choosing different tournament sizes. The benefits of applying this method are: accelerating the evolution process and paralleling the competition. It also eliminates the centralized fitness comparison among the individuals. We use Tournament Selection in our case studies.
- Breeding: The breeding process involves the crossover, mutation, and reproduction genetic operators, and starts after the selection algorithm is applied. The reproduction operator randomly selects an individual and sends a copy of that individual to the next generation. The crossover operation selects two individuals, randomly chooses a crossover point on each individual, exchanges the tree substructure below those points, and creates two new offspring individuals. The mutation operation chooses a random point in one selected individual, removes the tree substructure below that point, and inserts a randomly generated substructure. All offspring generated from crossover and mutation are sent to the next generation.
- Evolution: After breeding is complete, the new population replaces the old one. The *GP* process then repeats the evaluation, selection, and breeding steps on the new population. This evolutionary process continues until all terminating conditions are satisfied, i.e., the maximum number of generations, or a preferred solution is found. As the *GP* process evolves, the average fitness of the population is expected to be increased. The best individual of the given run is the solution found by *GP*.

The *GP* tool used in our study is "lilgp 1.01" developed by Douglas Zongker and Bill Punch of Michigan State University [29]. It is implemented in the C programming language, and is based on the LISP work of John Koza. When applying lilgp to a *GP* application, each individual is organized as an S-expression-tree where each node in the tree is a C function pointer.

4.2 Fitness Function

We consider two fitness functions during our *GP* modeling process: a primary fitness reflecting model performance and a secondary fitness representing model complexity. For a given data set, an individual's primary fitness is given by:

$$pFitness = \frac{N_I + cN_{II}}{N}, \tag{3}$$

where N_I is the number of Type_I errors, N_{II} is the number of Type_II errors, N is the total number of instances in the data set, and c is a modeling parameter (or weight) that can be varied to achieve the desired balance between the Type_I and Type_II error rates. The above equation indicates that a higher c value provides greater emphasis on correctly predicting the *fp* modules. The lower the $pFitness$ value is, the better the model performance is.

The secondary fitness function ($sFitness$) relates to the size of an individual, i.e., size of his S-expression-tree. Since simpler solutions and overfitting avoidance are closely associated [15], controlling code bloat is important and needs to be considered in our study [30], [31]. The secondary fitness function of an individual is defined as the number of nodes in its S-expression-tree. The fewer the nodes of an individual (i.e., smaller size), the better its fitness is. We select 10 as a size threshold to avoid losing too much diversity during the early generations of a *GP* run. If the number of tree nodes is less than 10, then the fitness of the model is assigned to 10.

TABLE 2
Model Overfitting Example

	KC3		MW1	
Model	Type_I	Type_II	Type_I	Type_II
1	15.70%	14.00%	46.50%	38.70%
2	16.40%	14.00%	22.30%	38.70%
3	13.30%	14.00%	44.60%	48.40%

We consider c values of 5, 8, 10, 15, and 20 to cover a broad range of models for obtaining a good representative of our model selection strategy. Based on preliminary investigations, it was found that a value of less than 5 for c tends to produce a model with very high Type_II error rates, i.e., most *fp* modules are misclassified as *nfp*. Moreover, it was found that a value of greater than 20 tends to produce a model with very high Type_I error rates, i.e., most *nfp* modules are misclassified as *fp*. We considered a very large number of c values between 5 and 20 (both inclusive), and found that the five values presented in the paper represented a good set of sample values for c. It was found that the other values of c which we considered in our experimentation did not provide a noticeable variation in the software quality models compared to those based on the five c values presented in the paper. However, another study with a different set of software projects would result in the selection of a different set of values for the modeling parameter c.

For a given *GP*-based software quality modeling task, we perform 100 independent runs. This consists of 20 runs, each with a different c value. Among the 100 *GP* runs, the top three models are selected and recorded. Among competing models with very similar error rates and preferred balance, a smaller model (size in number of tree nodes) is selected. The effectiveness of our fitness function strategy has been demonstrated in our prior work [19].

4.3 GP Multiple Data Sets Baseline Classifier

In this first strategy of *GP*-based software quality modeling with multiple software measurement data sets, among the seven data sets (see Section 3), one is selected as test data while the remaining six are combined to form the training or fit data. For the given training, the 100 *GP* runs are performed and the best three models are applied to the associated test data set. In order to avoid bias due to a lucky or unlucky *GP* run, an additional 100 runs are performed and the resulting additional three models are applied to the test data. The average test data performance of the six models is reported as the prediction capability. This modeling strategy is repeated such that all seven software measurement data sets are used as test data once.

4.4 GP Multiple Data Sets Validation Classifier

Similar to other search-based computational methods (e.g., tabu search, simulated annealing, neural network, etc.), *GP* is subject to closely emulating the training data rather than learning the underlying trends of the data, i.e., overfitting. The problem is compounded in *GP* with the uncertainty of knowing when overfitting will occur, making model selection a difficult problem. An example of this problem is presented in Table 2, which shows error rates of the three best models when KC3 is used as the training data and MW1 is used as the test data. The models that have similar Type_I and Type_II error rates on KC3 perform significantly differently (poorly in this case) on the test data, MW1.

The second strategy of *GP*-based software quality modeling with multiple data sets involves adding a validation phase for filtering models with poor generalization. This is in contrast to using only one validation data set [14], [15] which may not be effective in filtering out poor prediction. In this approach, for a given test data set, one of the six remaining data sets is used as training data, while the other five are used for validation. In the validation phase, models that have relatively similar performances on the validation data are selected as candidate models.

The initial pool of candidate models is comprised of those that perform well on the training data set. All of the candidate models are applied to each of the five validation data sets for evaluating their generalization capability. If a given model performs poorly on a validation data set, then it is removed from the list of candidates. Otherwise, the model remains as a candidate software quality model. A rule-of-thumb for defining poor performance on a validation data set is that if a model's Type_I or Type_II error rate is greater than 50 percent, then it is removed from the list of candidates. This process continues until all candidates have been evaluated on all five validation data sets. The resulting list of candidates is comprised of models with good generalization. The model that performs best on all five validation data sets is selected and applied to the test data set.

4.5 GP Multiple Data Sets Validation-and-Voting Classifier

Software quality modeling based on only one training data set may not be as effective as training on multiple data sets. This is likely because of a lower amount of software measurement and quality data available to the learner with just one data set compared to several [32]. The third strategy of *GP*-based software quality modeling with multiple data sets involves adding a voting phase during training with validation. While the validation phase focuses on the overfitting problem, adding a voting phase during training can alleviate the problem of training with just one data set.

Given a test data set and the remaining six data sets, the validation-and-voting (V-V) classifier considers each of the six data sets as a subsample of the collective training data set. The training and validation phases are repeated six times, each with a different training subsample, and consequently, six final models are generated. The predictions of the test data program modules are obtained by polling the six models based on the following voting strategy: If three or more models predict a module as *fp*, then it is predicted as *fp*; otherwise, it is predicted as *nfp*. This simple validation-and-voting strategy results in obtaining a robust software quality model that has learned from multiple data sets with improved predictions.

5 Case Study 2 Methodology

It is shown in Section 6.1 that the *GP*-based V-V approach yielded the best results among the three strategies presented in Case Study 1. The aim of conducting this second

TABLE 3
Seventeen Learners of Case Study 2

Classification Technique	Acronym
Locally Weighted Learning (with Decision Stump) [33]	LWLStump
1-Instance Based Learning [32]	IB1
k-Instance Based Learning [8]	IBk
Bagging [34]	Bagging
Sequential Minimal Optimization [35] (Support Vector Machine)	SMO
Logistic Regression [36]	LR
Ripple Down Rules [37]	Ridor
One Rule [38]	OneR
Lines-of-Code	LOC
Decision Table [39]	DecisionTable
WEKA's implementation of C4.5 [40]	J48
Partial Decision Tree [41]	PART
Tree-Disc Classification Tree [42]	TD
Alternating Decision Tree [43]	ADTree
Repeated Incremental Pruning to Produce Error Reduction [44]	JRip
Random Forest [32]	RandomForest
Naive Bayes [45]	NaiveBayes

case study was to compare the *GP*-based V-V approach with a similar voting approach but with non-*GP* classifiers. Such a comparison provides practical merit to a practitioner interested in adopting software quality modeling with multiple software data repositories.

This case study involves 17 different machine learners (nonevolutionary techniques) applied to the same seven software measurement data sets used in Case Study 1 [16]. The 17 learners are listed in Table 3, which also provides relevant references for the reader. A detailed description of those learners is out-of-scope of this paper; however, the reader can find those details in our prior work [23] and/or in the WEKA data mining tool [32].

A given classification technique is trained on all seven software data sets individually, resulting in seven models for the given technique. For a given data set considered as test data, the remaining six models are applied to predict the fault-proneness of modules in the test data set, resulting in six prediction vectors for the test data. Similarly to the *GP*-based V-V approach of Case Study 1, a voting among the six predictions determines the final fault-proneness prediction for the test data program modules. More specifically, if three or more models predict a module as *fp*, then it is labeled as *fp*, and *nfp* otherwise. The model training and voting process is repeated for all 17 learners, and instead of presenting the results of each learner for each test data set, we summarize the performance of all of the classifiers for each test data set.

6 Results and Analysis

6.1 Case Study 1: Results

We set the various *GP* modeling parameters required by lilgp [29] as shown below:

- pop_size: 1,000
- max_generations: 50
- random_seed: 3
- output.basename: cccs1

TABLE 4
Baseline Classifier—Training Data Results

Fit data	Type_I	Type_II	Overall
not-KC1	33.61%	30.09%	33.02%
not-KC2	32.21%	30.87%	31.99%
not-KC3	32.19%	31.40%	32.06%
not-CM1	33.32%	30.23%	32.80%
not-MW1	34.11%	29.02%	33.26%
not-PC1	36.14%	27.02%	34.55%
not-JM1	24.39%	28.99%	24.96%

- output.bestn: 1
- init.method: half_and_half
- init.depth: 2-6
- max_depth: 20
- breed_phases: 3
- breed[1].operator: crossover, sele=tournament
- breed[1].rate: 0.60
- breed[2].operator: reproduction, sele=tournament
- breed[2].rate: 0.10
- breed[3].operator: mutation, sele=tournament
- breed[3].rate: 0.30
- function set: +, –, *, /, sin, cos, exp, log, GT, VGT.

Most of the parameters are fixed except for output.basename and random_seed in our *GP* experiments. The parameter output.basename defines a file name to store the results of a *GP* run and random_seed provides a random starting point for each *GP* process. The two operators, GT and VGT, create discontinuous functions for a model. Operator GT returns 0.0 if the value of the first parameter is greater than the second one; otherwise, it returns 1.0. Operator VGT returns the maximum of the two arguments. The maximum number of generations for a *GP* run is set to 50, except in the case of the Baseline Classifier, where the merged training data set is relatively larger than an individual data set. Optimization of the *GP* modeling parameters is out-of-scope of this paper; however, specific values are selected based on suggestions in related literature and experience from our prior work in genetic programming [19].

6.1.1 Multiple Data Sets Baseline Classifier

The *GP*-based Baseline Classifier results for the different merged training data sets are shown in Table 4, which shows the Type_I, Type_II, and Overall misclassification rates. The notations of the first column indicate which of the seven data sets was not included to form the merged training data set, e.g., not-KC1 represents a training data set formed by merging program modules from the KC2, KC3, CM1, MW1, PC1, and JM1 data sets, while KC1 is treated as the test data. The error rates shown are averages of the six best models obtained from 200 *GP* runs, as explained earlier. The relative balance between the Type_I and Type_II error rates is reflective of our model selection strategy explained in Section 2. The results of not-JM1 are relatively better than other trained models, which is indicative of data noise present in the JM1 data set [23].

The different Baseline Classifiers were applied to their respective test data sets, and those results are summarized in Table 5. Once again, the three types of misclassification

TABLE 5
Baseline Classifier—Test Data Results

Test data	Type_I	Type_II	Overall
KC1	28.24%	31.79%	28.79%
KC2	25.52%	21.23%	24.65%
KC3	28.39%	24.03%	27.98%
CM1	51.50%	23.96%	48.88%
MW1	41.76%	24.19%	40.41%
PC1	39.27%	20.39%	37.97%
JM1	22.49%	49.50%	27.63%

rates are presented for each test data set. The KC1, KC2, and KC3 results are relatively similar to their corresponding training data performances. However, in the case of CM1, MW1, PC1, and JM1, there is considerable discrepancy between their test data performances and training data performances (Table 4).

6.1.2 Multiple Data Sets Validation Classifier

The second *GP*-based strategy for software quality modeling with multiple data sets is based on adding a validation phase during model training. Instead of training with a merged data set, only one of the six data sets is used for training, while the other five are used for filtering out overfitted models. An example of how validation is performed is presented in Table 6, which corresponds to when PC1 is used as test data and KC1 is used for training. The column headings "M_i" indicates the best model obtained from the ith run, e.g., M_3 implies that the model was generated by the third run, while M_6 implies that the model was generated by the sixth run.

The table lists three flags for the different *GP* models based on their performance on the five validation data sets: "p" (*pass*), "f" (*fail*), and "-" (*model removed*). If a model has good performance on a given validation data set, then a "p" is assigned to that model and it is marked as a potential candidate. On the other hand, if a model has poor performance on a given validation data set, i.e., shows overfitting, then an "f" is assigned to that model and it is removed from the list of potential candidates. Once a model is removed, it is not tested on the remaining validation data sets. This implies that if a model has been removed from the list of potential candidates, it is marked with an "-" in the table.

Referring to example shown in Table 6, after training with KC1 is completed, six models, M_3, M_6, M_7, M_15, M_16, and M_17, were identified as potential candidates. During the validation process, M_6 was removed as a candidate since it had poor performance on KC2. Similarly, M_3, M_15, and M_17 were removed since they had poor performances on MW1. Upon completing the validation process, M_7 and M_16 remain as potential candidates. A comparison of these two models on all five validation data sets resulted in selecting M_16 as the final model. Hence, when KC1 is used for training, M_16 is the best model which is then applied to the test data set, PC1.

TABLE 6
Multiple Data Sets Validation—PC1 Test and KC1 Fit

Data	M_3	M_6	M_7	M_15	M_16	M_17
KC2	p	f	p	p	p	p
KC3	p	–	p	p	p	p
CM1	p	–	p	p	p	p
MW1	f	–	p	f	p	f
JM1	–	–	p	–	p	–

TABLE 7
Validation Classifier—Test Data Results

Test data	Type_I	Type_II	Overall
KC1	30.30%	33.85%	30.85%
KC2	25.97%	27.36%	26.25%
KC3	30.56%	32.17%	30.71%
CM1	29.80%	32.08%	30.02%
MW1	40.81%	25.16%	39.60%
PC1	38.27%	32.11%	37.85%
JM1	31.52%	46.26%	34.33%
Average	32.46%	32.71%	32.80%

For a given test data set, the process of selecting one of the remaining six data sets as fit data and using the other five for validation is repeated six times, i.e., each of the remaining six data sets is used once for model training. This results in six validated models obtained corresponding to a given test data set. We report the average performances of these six models for a given test data set. Those results are summarized in Table 7, which again shows the Type_I, Type_II, and Overall misclassification rates.

6.1.3 Multiple Data Sets Validation-and-Voting Classifier

The third *GP*-based strategy for software quality modeling with multiple data sets extends the Validation Classifier by incorporating a voting scheme. The V-V Classifier polls the test data predictions of the six best models obtained after validation, as explained in Section 4. The polled test data prediction results are summarized in Table 8. We note a general reduction (compared to Baseline Classifier and Validation Classifier) in the Type_II error rates for the test data sets with a relatively similar or minimally-higher Type_I error rates.

In addition to the predicted numbers of *fp* and *nfp* program modules, a practitioner is also interested in knowing the total cost of misclassification at a cost ratio relevant to his software project. We compare the three *GP* approaches based on three cost ratio values in addition to their misclassification error rates. We consider $\frac{C_{II}}{C_I}$ values of 15, 20, and 25 as these values are more representative of

TABLE 8
Validation and Voting Classifier—Test Data Results

Test data	Type_I	Type_II	Overall
KC1	35.19%	21.54%	33.08%
KC2	30.19%	13.21%	26.73%
KC3	35.90%	13.95%	33.84%
CM1	31.73%	20.83%	30.69%
MW1	46.24%	22.58%	44.42%
PC1	42.19%	19.74%	40.65%
JM1	35.46%	35.51%	35.47%
Average	36.70%	21.05%	34.98%

TABLE 9
Comparing *GP* Classifiers on KC1 Test Data

Classifier	Error Rates		$\frac{C_{II}}{C_I}$		
	Type_I	Type_II	15	20	25
Baseline	28.24%	31.79%	0.97	1.22	1.46
Validation	30.30%	33.85%	1.04	1.30	1.56
V-V	35.19%	21.54%	0.80	0.96	1.13

TABLE 10
Comparing *GP* Classifiers on KC2 Test Data

Classifier	Error Rates		$\frac{C_{II}}{C_I}$		
	Type_I	Type_II	15	20	25
Baseline	25.50%	21.23%	0.85	1.07	1.28
Validation	25.97%	27.36%	1.04	1.32	1.60
V-V	30.19%	13.21%	0.64	0.78	0.91

TABLE 11
Comparing *GP* Classifiers on KC3 Test Data

Classifier	Error Rates		$\frac{C_{II}}{C_I}$		
	Type_I	Type_II	15	20	25
Baseline	28.39%	24.03%	0.60	0.71	0.82
Validation	30.56%	32.17%	0.73	0.88	1.03
V-V	35.90%	13.95%	0.52	0.59	0.65

TABLE 12
Comparing *GP* Classifiers on CM1 Test Data

Classifier	Error Rates		$\frac{C_{II}}{C_I}$		
	Type_I	Type_II	15	20	25
Baseline	51.50%	23.96%	0.81	0.92	1.04
Validation	29.80%	32.08%	0.73	0.88	1.03
V-V	31.73%	20.83%	0.58	0.68	0.78

TABLE 13
Comparing *GP* Classifiers on MW1 Test Data

Classifier	Error Rates		$\frac{C_{II}}{C_I}$		
	Type_I	Type_II	15	20	25
Baseline	41.76%	24.19%	0.66	0.76	0.85
Validation	40.81%	25.16%	0.67	0.76	0.86
V-V	46.24%	22.58%	0.69	0.77	0.86

high-assurance systems such as those considered in our case studies [8]. A practitioner can gain insight into which approach performs better at a cost ratio that is closer to his target value.

The relative comparison of the three approaches is presented in Tables 9, 10, 11, 12, 13, 14, and 15 for the seven test data sets. These tables list the respective misclassification rates and *NECM* values for the three *GP*-based approaches. Comparison based solely on Type_I and Type_II error rates is rather complicated since the two are inversely proportional. Among two competing models, one can have a higher Type_I rate but lower Type_II rate while the other can have a much lower Type_I rate but higher Type_II rate. Computing the *NECM* values makes this comparison task a bit simpler. Generally, a lower value of *NECM* indicates lower misclassification costs, and therefore, a better software quality model.

TABLE 14
Comparing *GP* Classifiers on PC1 Test Data

Classifier	Error Rates		$\frac{C_{II}}{C_I}$		
	Type_I	Type_II	15	20	25
Baseline	38.96%	21.93%	0.59	0.66	0.74
Validation	38.27%	32.11%	0.69	0.80	0.91
V-V	42.19%	19.74%	0.60	0.66	0.73

TABLE 15
Comparing *GP* Classifiers on JM1 Test Data

Classifier	Error Rates		$\frac{C_{II}}{C_I}$		
	Type_I	Type_II	15	20	25
Baseline	22.49%	49.50%	1.60	2.07	2.54
Validation	31.52%	46.26%	1.58	2.02	2.46
V-V	35.50%	35.50%	1.30	1.64	1.98

TABLE 16
V-V versus Baseline—Difference Statistics

Metric	$\frac{C_{II}}{C_I}$		
	15	20	25
Mean	0.136	0.190	0.241
Stdev	0.125	0.161	0.203
Highest	0.300	0.430	0.560
Lowest	-0.030	-0.010	-0.010
Median	0.170	0.240	0.260
AAD	0.100	0.124	0.156
p-value	0.029	0.021	0.020

In the case of test data sets KC1, KC2, KC3, CM1, and JM1, the Validation-and-Voting Classifier yielded the lowest total cost of misclassification compared to the Baseline Classifier and Validation Classifier. This is true for all three cost ratios considered in our comparison. When MW1 is the test data set, the three approaches are relatively similar in terms of *NECM* values for all three cost ratios. This suggests that, for MW1, the Baseline Classifier cannot be improved much by introducing the validation and/or voting phases during software quality modeling. A similar conclusion can be made when PC1 is the test data set. More specifically, there is relatively no difference in total misclassification costs of the Baseline Classifier and V-V Classifier.

A statistical analysis was performed to evaluate the significance of performance differences between the Baseline Classifier and the Validation-and-Voting Classifier. A t-test is conducted to evaluate this significance with the hypothesis that the V-V Classifier is better than the Baseline Classifier in terms of *NECM* values. The statistical results for the differences between the paired data of each test data are summarized in Table 16. The table lists the mean, standard deviation, maximum, minimum, median, average absolute deviation, and p-value. The very low p-values (less than 0.05) for all three cost ratios clearly indicate that the V-V classifier is significantly better than the Baseline Classifier at the 95 percent confidence interval.

6.2 Case Study 2 Results

The average performances of the 17 classifiers for each training data set are shown in Table 17. Recall that each of the 17 learners is trained on each of the seven software

TABLE 17
Case Study 2: Fit Data Performance

Fit data	Type_I	Type_II	Overall
KC1	27.7%	27.1%	27.6%
KC2	21.7%	20.1%	21.3%
KC3	25.5%	23.8%	25.4%
CM1	30.2%	24.6%	29.7%
MW1	31.2%	30.2%	31.2%
PC1	23.0%	22.5%	23.0%
JM1	33.6%	33.1%	33.5%
Average	27.9%	26.2%	27.7%

TABLE 19
Case Study 2: Test Data Performance

Test data	Error Rates		$\frac{C_{II}}{C_I}$		
	Type_I	Type_II	15	20	25
KC1	28.4%	29.8%	0.93	1.16	1.39
KC2	29.9%	18.5%	0.80	0.99	1.18
KC3	29.6%	23.0%	0.59	0.70	0.81
CM1	32.1%	30.1%	0.72	0.86	1.01
MW1	36.6%	25.4%	0.63	0.73	0.83
PC1	34.6%	29.7%	0.63	0.73	0.83
JM1	30.9%	41.6%	1.44	1.84	2.24

measurement data sets. We present the error rates averaged across all 17 learners instead of presenting details on all 17 learners individually. The error rates for each learner averaged across all seven training data sets are presented in Table 18. Similarly to Case Study 1, the relative balance between the Type_I and Type_II error rates is reflective of our model selection strategy. Among the 17 learners, TD provides the best error rates, while IB1, DecisionTable, NaiveBayes, and LOC provide the worst error rates. Random Forest, Logistic Regression, and IBk learners provide competitive fault-proneness predictions.

The test data predictions are obtained based on the voting approach explained in Section 5, which is the same approach used by the *GP*-based Validation-and-Voting approach. This similarity of voting approaches allows us to perform a direct comparison between the two case studies. The test data performances of the 17 learners averaged for each test data set are summarized in Table 19. The table shows the three types of misclassification error rates and the (normalized) total cost of misclassification associated with each test data set. Compared to the other data sets, JM1 has higher *NECM* values, which reflects the higher level of noise present in the JM1 data set [23]. A similar deduction can be made for KC1.

6.3 Case Study Comparisons

The two case studies presented software quality modeling with multiple software measurement data sets, and evaluated the various models based on total expected cost of misclassification. In Case Study 1, the Validation and Voting strategy was deemed best among the three competing approaches for modeling with multiple data sets. Case Study 2 considers non-*GP* machine learners for software quality modeling with the same seven software measurement data sets. We compare results of the V-V approach with those of Case Study 2.

The expected cost of misclassification values for both approaches is summarized in Table 20. The two approaches are compared with respect to the same cost ratio values, i.e., 15, 20, and 25. A quick comparison of the respective averages across all test data sets suggests that the *GP*-based software quality models yield lower *NECM* values compared to the non-*GP* models. This is true for all three cost ratio values. A more detailed look at the individual test data sets indicates that the *GP*-based models consistently yield lower *NECM* values compared to non-search-based software quality models. While a statistical comparison would reveal the significance of this improvement, it is clear that investing in a search-based approach for software quality modeling with multiple data sets is worth the effort. Given the substantial improvement in software quality prediction, the relatively higher runtime complexity of the GP-based approach can be justified.

A likely reason why the GP-based models performed better in this study is that the search space for the solutions is more widely explored by GP as compared to the more traditional classification algorithms which generally make a strong assumption about the structure of the software quality model. The S-expression solution of GP has more freedom in forming the structure of the final solution as compared to the more traditional approaches, such as

TABLE 18
Case Study 2: Performance of 17 Classifiers

Fit data	Type_I	Type_II	Overall
J48	27.0%	26.1%	26.9%
JRip	26.9%	24.4%	26.6%
NaiveBayes	29.0%	29.4%	29.0%
DecisionTable	29.4%	28.7%	29.3%
RandomForest	25.6%	24.1%	25.4%
OneR	26.6%	25.8%	26.5%
PART	26.4%	25.2%	26.2%
IBk	25.4%	24.8%	25.3%
IB1	30.8%	27.8%	30.4%
ADTree	26.9%	27.0%	26.9%
Ridor	36.6%	24.5%	35.3%
LWLStump	26.9%	26.8%	26.9%
SMO	26.2%	26.7%	26.3%
Bagging	26.9%	24.4%	26.5%
LOC	31.5%	28.8%	31.1%
LR	25.9%	25.5%	25.8%
TD	21.2%	20.5%	21.0%
Average	27.9%	26.2%	27.7%

TABLE 20
Case Study 1 versus Case Study 2

Test data	Case Study One $\frac{C_{II}}{C_I}$			Case Study Two $\frac{C_{II}}{C_I}$		
	15	20	25	15	20	25
KC1	0.80	0.96	1.13	0.93	1.16	1.39
KC2	0.64	0.78	0.91	0.80	0.99	1.18
KC3	0.52	0.59	0.65	0.59	0.70	0.81
CM1	0.58	0.68	0.78	0.72	0.86	1.01
MW1	0.69	0.77	0.86	0.63	0.73	0.83
PC1	0.60	0.66	0.73	0.63	0.73	0.83
JM1	1.30	1.64	1.98	1.44	1.84	2.24
Average	0.73	0.87	1.01	0.82	1.00	1.18

Decision Trees or Naive Bayes. The GP-based approach makes use of the unique advantage of GP, namely, 1) it performs a much larger search of the solution space for the given problem and 2) different models can be produced during the GP search process even though all of the parameters are kept unchanged during the modeling process. It is important to note that, from a practical point of view, the application of a complex search technique, such as GP, must be justifiable given the complexity of the search problem. A runtime analysis of a search-based solution can provide insight into the justification of its usage [46], [47].

6.4 Threats to Empirical Validity

Due to the many human factors that affect software development, and consequently, software quality, controlled experiments for evaluating the usefulness of empirical models are not practical. We adopted a case study approach in the empirical investigations presented in this paper. To be credible, the software engineering community demands that the subject of an empirical study have the following characteristics [48]:

- Developed by a group, and not by an individual.
- Be as large as industry-size projects, and not a toy problem.
- Developed by professionals, and not by students.
- Developed in an industry/government organization setting, and not in a laboratory.

We note that our case studies fulfill all of the above criteria. The software systems investigated in our study were developed by professionals in a government software development organization. In addition, each system was developed to address a real-world problem.

Empirical studies that evaluate measurements and models across multiple projects should take care in assessing the scope and impact of its analysis and conclusion. For example, C and C++ projects may be considered similar if they are developed using the procedural paradigm. Combining object-oriented project data (e.g., Java) with non-object-oriented project data (e.g., C or C++) needs careful consideration. For example, per module lines of code of an object-oriented software tends to be lower than that of a non-object-oriented software.

The proposed process of combining multiple learners and data sets for software quality analysis included four C projects, two C++ projects, and one Java project. We introduced two additional metrics: one to capture data-set-size variation and another to capture whether a module belongs to an OO project. The average Loc_Total of the C projects was relatively similar to that of the C++ projects. The average Loc_Total for the Java project, while slightly lower, was relatively comparable. All data sets were normalized and scaled to account for variation in data set size.

The use of other software metrics (OO and process metrics) may improve the outcome of software quality modeling, and further ratify the basic conclusions derived in our study. However, we believe that, in the case of the seven projects used in our study, a similarity of application domain and software development process would outweigh any adverse impact the OO-languages would have in our multidata set combination study.

In addition to commonality of development organization and application domain, all projects were characterized by the same set of metrics. The similarity of projects emphasized in our paper applies to a similarity in their development organization and application domain. All software projects used in our study were developed under NASA software development organization process, and all pertain to mission-critical software applications. Such similarity among the projects was considered sufficient for the primary scope of our study, i.e., improving software quality analysis by evaluating multiple software project repositories.

As with any work in empirical software engineering, the specific results of this work are based on the case study data examined during empirical analysis. It is important to know that practical issues related to data collection, such as inconsistent data interpretation, erroneous data recording, implausible values, etc., may affect case study outcomes. Other than JM1, the seven software measurement data sets obtained from the NASA Metrics Data Program were used (except for normalization and scaling) as is. The JM1 data set was known to have some obvious data noise based on our prior work [23], and hence, was cleansed prior to modeling. No other changes were made to the case study data other than those stated in Section 3, allowing for easier and comparable replication studies.

In our study, the evolutionary process of a typical GP run is conducted with a selected setting for the different GP parameters, including crossover rate, mutation rate, population, etc. We note that it is likely that the obtained results could be improved by optimizing these GP parameters. However, such an analysis was out of the scope of this study.

7 Conclusion

A focus on search-based software quality modeling with multiple software data repositories is presented in this paper. Traditionally, predictive software quality models are obtained by learning from one software measurement and defect data set of a prior release or similar project. However, the knowledge obtained from just one training data set may not sufficiently learn the software quality and defect occurrence trends of the development organization. We hypothesize that optimizing software quality models with the help of multiple software projects can improve upon the predictive capability of a one data set software quality model.

A genetic-programming-based approach is taken for building optimal software quality models from multiple data sets. Three different *GP*-based strategies are presented for that purpose: Baseline Classifier, Validation Classifier, and Validation-and-Voting Classifier. Software measurement and defect data from seven real-world high-assurance systems are used for demonstrating the different modeling approaches. It is shown that the Validation-and-Voting Classifier is generally better than the Baseline Classifier, and is less prone to overfitting and predictive variance.

A second case study focuses on the same problem, i.e., software quality modeling with multiple data sets, but involves 17 different machine learning algorithms—all non-search-based learners. The same seven software measurement data sets are used in this case study. A comparison between the two case studies clearly indicates that the

search-based software quality modeling approach provides a better solution when modeling from multiple software data sets. However, in the case of software quality engineering and as demonstrated by this study, it is advised to use software data repositories of existing projects similar to the target project in terms of software quality requirements, measurements, and application domain.

Future works will investigate the use of learners trained on different representations of the same input, e.g., investigate whether the inclusions of object-oriented metrics, process, and other measurements can improve the performance of software quality models presented in this paper. In addition, one could consider examining more sophisticated voting schemes, such as weighted voting or cascading. Such schemes may improve the predictive accuracy of the single-learner and multidata set software quality model.

ACKNOWLEDGMENTS

The authors are grateful to the guest editors of the special section on Search-Based Optimization for Software Engineering, Professor Harman and Dr. Mansouri, and the four anonymous reviewers for their constructive criticism and suggestions which went toward improving this paper. They thank the various members of the Empirical Software Engineering Laboratory and Data Mining and Machine Learning Laboratory at Florida Atlantic University for their reviews of this paper. They are grateful to the staff of the NASA Metrics Data Program for making the software measurement data available. The work of Yi Liu was supported in part by a summer research grant from The J. Whitney Bunting School of Business at Georgia College and State University, Milledgeville.

REFERENCES

[1] N.F. Schneidewind, "Body of Knowledge for Software Quality Measurement," *Computer,* vol. 35, no. 2, pp. 77-83, Feb. 2002.
[2] L.C. Briand, W.L. Melo, and J. Wust, "Assessing the Applicability of Fault-Proneness Models across Object-Oriented Software Projects," *IEEE Trans. Software Eng.,* vol. 28, no. 7, pp. 706-720, July 2002.
[3] N.J. Pizzi, R. Summers, and W. Pedrycz, "Software Quality Prediction Using Median-Adjusted Class Labels," *Proc. IEEE CS Int'l Joint Conf. Neural Networks,* vol. 3., pp. 2405-2409, May 2002.
[4] A. Koru and H. Liu, "Building Effective Defect-Prediction Models in Practice," *IEEE Software,* vol. 22, no. 6, pp. 23-29, Nov./Dec. 2005.
[5] N.F. Schneidewind, "Investigation of Logistic Regression as a Discriminant of Software Quality," *Proc. IEEE CS Seventh Int'l Software Metrics Symp.,* pp. 328-337, Apr. 2001.
[6] T. Menzies, J. Greenwald, and A. Frank, "Data Mining Static Code Attributes to Learn Defect Predictors," *IEEE Trans. Software Eng.,* vol. 33, no. 1, pp. 2-13, Jan. 2007.
[7] L. Guo, B. Cukic, and H. Singh, "Predicting Fault Prone Modules by the Dempster-Shafer Belief Networks," *Proc. IEEE CS 18th Int'l Conf. Automated Software Eng.,* pp. 249-252, Oct. 2003.
[8] T.M. Khoshgoftaar and N. Seliya, "Comparative Assessment of Software Quality Classification Techniques: An Empirical Case Study," *Empirical Software Eng. J.,* vol. 9, no. 3, pp. 229-257, 2004.
[9] N.E. Fenton and S.L. Pfleeger, *Software Metrics: A Rigorous and Practical Approach,* second ed. PWS Publishing, 1997.
[10] M. Harman, "The Current State and Future of Search Based Software Engineering," *Proc. IEEE CS Workshop Future of Software Eng.,* pp. 342-357, May 2007.
[11] M. Harman and B. Jones, "Search Based Software Engineering," *J. Information and Software Technology,* vol. 43, no. 14, pp. 833-839, 2001.
[12] J.R. Koza, *Genetic Programming,* vol. 1. MIT Press, 1992.
[13] T.M. Mitchell, *Machine Learning.* McGraw-Hill, 1997.
[14] I. Kushchu, "Genetic Programming and Evolutionary Generalization," *IEEE Trans. Evolutionary Computation,* vol. 6, no. 5, pp. 431-442, Oct. 2002.
[15] C. Gagné, M. Schoenauer, M. Parizeau, and M. Tomassini, "Genetic Programming, Validation Sets, and Parsimony Pressure," *Proc. Ninth European Conf. Genetic Programming,* P. Collet, M. Tomassini, M. Ebner, S. Gustafson, and A. Ekárt, eds., pp. 109-120, Springer, Apr. 2006.
[16] T.M. Khoshgoftaar, P. Rebours, and N. Seliya, "Software Quality Analysis by Combining Multiple Projects and Learners," *Software Quality J.,* vol. 17, no. 1, pp. 25-49, Mar. 2009.
[17] M.J. Meulen and M.A. Revilla, "Correlations between Internal Software Metrics and Software Dependability in a Large Population of Small C/C++ Programs" *Proc. 18th IEEE Int'l Symp. Software Reliability Eng.,* pp. 203-208, Nov. 2007.
[18] C. Kaner and W.P. Bond, "Software Engineering Metrics: What Do They Measure and How Do We Know," *Proc. 10th IEEE Int'l Software Metrics Symp.,* Sept. 2004.
[19] T.M. Khoshgoftaar and Y. Liu, "A Multi-Objective Software Quality Classification Model Using Genetic Programming," *IEEE Trans. Reliability,* vol. 56, no. 2, pp. 237-245, June 2007.
[20] T.M. Khoshgoftaar, N. Seliya, and D.D. Drown, "On the Rarity of Fault-Prone Modules in Knowledge-Based Software Quality Modeling," *Proc. 20th Int'l Conf. Software Eng. and Knowledge Eng.,* July 2008.
[21] A. Folleco, T.M. Khoshgoftaar, J. VanHulse, and L. Bullard, "Software Quality Modeling: The Impact of Class Noise on the Random Forest Classifier," *Proc. IEEE World Congress on Computational Intelligence,* June 2008.
[22] S. Lessmann, B. Baesens, C. Mues, and S. Pietsch, "Benchmarking Classification Models for Software Defect Prediction: A Proposed Framework and Novel Findings," *IEEE Trans. Software Eng.,* vol. 34, no. 4, pp. 485-496, July/Aug. 2008.
[23] T.M. Khoshgoftaar and N. Seliya, "The Necessity of Assuring Quality in Software Measurement Data," *Proc. IEEE CS 10th Int'l Symp. Software Metrics,* pp. 119-130, Sept. 2004.
[24] J. Sayyad Shirabad and T. Menzies, "The PROMISE Repository of Software Engineering Databases." School of Information Technology and Eng., Univ. of Ottawa, http://promise.site.uottawa.ca/SERepository, 2005.
[25] J.J. Cuadrado-Gallego, L. Fernández-Sanz, and M.-Á. Sicilia, "Enhancing Input Value Selection in Parametric Software Cost Estimation Models through Second Level Cost Drivers," *Software Quality J.,* vol. 14, no. 4, pp. 330-357, Dec. 2006.
[26] M. Shepperd and G. Kadoda, "Comparing Software Prediction Techniques Using Simulation," *IEEE Trans. Software Eng.,* vol. 27, no. 11, pp. 1014-1022, Nov. 2001.
[27] K. Sunghun, T. Zimmermann, E.J. Whitehead, and A. Zeller, "Predicting Faults from Cached History," *Proc. 29th Int'l Conf. Software Eng.,* pp. 489-498, 2007.
[28] W. Banzhaf, P. Nordin, R.E. Keller, and F.D. Francone, *Genetic Programming: An Introduction on the Automatic Evolution of Computer Programs and Its Application.* PWS Publishing Company, 1998.
[29] GP-Tool, http://garage.cse.msu.edu/software/lil-gp/, 1998.
[30] H. Iba, H. de Garis, and T. Sato, "Genetic Programming Using Minimum Description Length Principle," *Advances in Genetic Programming: Complex Adaptive Systems,* pp. 265-284, MIT Press, 1994.
[31] B.T. Zhang and H. Muhlenbein, "Balancing Accuracy and Parsimony in Genetic Programming," *Evolutionary Computation,* vol. 3, no. 1, pp. 17-38, 1995.
[32] I.H. Witten and E. Frank, *Data Mining: Practical Machine Learning Tools and Techniques,* second ed. Morgan Kaufmann, 2005.
[33] C.G. Atkeson, A.W. Moore, and S. Schaal, "Locally Weighted Learning," *Artificial Intelligence Rev.,* vol. 11, nos. 1-5, pp. 11-73, 1997.
[34] L. Breiman, "Bagging Predictors," *Machine Learning,* vol. 24, no. 2, pp. 123-140, 1996.
[35] J.C. Platt, "Sequential Minimal Optimization: A Fast Algorithm for Training Support Vector Machines," Technical Report 98-14, Microsoft Research, Apr. 1998.
[36] T.M. Khoshgoftaar and E.B. Allen, "Logistic Regression Modeling of Software Quality," *Int'l J. Reliability, Quality, and Safety Eng.,* vol. 6, no. 4, pp. 303-317, 1999.

[37] B.R. Gaines and P. Compton, "Induction of Ripple-Down Rules Applied to Modeling Large Databases," *J. Intelligent Information Systems*, vol. 5, no. 3, pp. 211-228, 1995.
[38] R.C. Holte, "Very Simple Classification Rules Perform Well on Most Commonly Used Data Sets," *Machine Learning*, vol. 11, pp. 63-91, 1993.
[39] R. Kohavi, "The Power of Decision Tables," *Proc. European Conf. Machine Learning*, N. Lavrač and S. Wrobel, eds., pp. 174-189, 1995.
[40] J.R. Quinlan, *C4.5: Programs for Machine Learning*. Morgan Kaufmann, 1993.
[41] E. Frank and I.H. Witten, "Generating Accurate Rule Sets without Global Optimization," *Proc. 15th Int'l Conf. Machine Learning*. pp. 144-151, 1998.
[42] T.M. Khoshgoftaar, X. Yuan, and E.B. Allen, "Balancing Misclassification Rates in Classification Tree Models of Software Quality," *Empirical Software Eng.*, vol. 5, pp. 313-330, 2000.
[43] Y. Freund and L. Mason, "The Alternating Decision Tree Learning Algorithm," *Proc. 16th Int'l Conf. Machine Learning*, pp. 124-133, 1999.
[44] W.W. Cohen, "Fast Effective Rule Induction," *Proc. 16th Int'l Conf. Machine Learning*, A. Prieditis and S. Russell, eds., pp. 115-123, July 1995.
[45] E. Frank, L. Trigg, G. Holmes, and I.H. Witten, "Naive Bayes for Regression," *Machine Learning*, vol. 41, no. 1, pp. 5-25, 2000.
[46] A. Arcuri, P.K. Lehre, and X. Yao, "Theoretical Runtime Analyses of Search Algorithms on the Test Data Generation for the Triangle Classification Problem," *Proc. IEEE CS First Int'l Workshop Search-Based Software Testing in Conjunction with ICST '08*, pp. 161-169, Apr. 2008.
[47] A. Arcuri, P.K. Lehre, and X. Yao, "Theoretical Runtime Analysis in Search Based Software Engineering," Technical Report CSR-09-04, Univ. of Birmingham, ftp://ftp.cs.bham.ac.uk/pub/tech-reports/2009/CSR-09-04.pdf, 2009.
[48] C. Wohlin, P. Runeson, M. Host, M.C. Ohlsson, B. Regnell, and A. Wesslen, *Experimentation in Software Engineering: An Introduction*. Kluwer Academic Publishers, 2000.

Yi (Cathy) Liu received the PhD degree in computer science from the Department of Computer Science and Engineering at Florida Atlantic University in 2003. She is currently an associate professor of computer science at Georgia College and State University. Her research interests include software engineering, software metrics, software reliability and quality engineering, computer performance modeling, genetic programming, and data mining. She is a member of the IEEE Conputer Society.

Taghi M. Khoshgoftaar is a professor in the Department of Computer Science and Engineering at Florida Atlantic University and the Director of the Empirical Software Engineering Laboratory and Data Mining and Machine Learning Laboratory. His research interests are in software engineering, software metrics, software reliability and quality engineering, computational intelligence, computer performance evaluation, data mining, machine learning, and statistical modeling. He has published more than 400 refereed papers in these areas. He is a member of the IEEE, the IEEE Computer Society, and the IEEE Reliability Society. He was the program chair and general chair of the IEEE International Conference on Tools with Artificial Intelligence in 2004 and 2005, respectively, and was the program chair of the 20th International Conference on Software Engineering and Knowledge Engineering in 2008. He was the general chair of the 21st International Conference on Software Engineering and Knowledge Engineering in 2009. He has served on technical program committees of various international conferences, symposia, and workshops. Also, he has served as the North American editor of the *Software Quality Journal*, and was on the editorial boards of the journals *Multimedia Tools and Applications* and *Empirical Software Engineering* and is on the editorial boards of the journals *Software Quality*, *Fuzzy Systems*, and *Knowledge and Information Systems*.

Naeem Seliya received the PhD degree in computer engineering from Florida Atlantic University, Boca Raton, in 2005. He is currently an assistant professor of computer and information science at the University of Michigan–Dearborn. His research interests include software engineering, data mining, machine learning, application and data security, computer forensics, and medical informatics. He is a member of the IEEE and the IEEE Computer Society.