Software Quality Modeling and Estimation with Missing Data

NAEEM SELIYA¹, TAGHI M. KHOSHGOFTAAR²*

¹Computer and Information Science, University of Michigan – Dearborn
4901 Evergreen Road, Dearborn, MI 48128, USA, January 2008

²Computer Science and Engineering, Florida Atlantic University
777 Glades Road, Boca Raton, FL 33431, USA.

(Received on March 15, 2007)

Abstract: Software quality estimation models generally exploit the software engineering measurements hypothesis that software metrics encapsulate the underlying quality of the software system. A typical model is trained using software measurements and fault data of a similar, previously developed project. Such a strategy requires complete knowledge of fault data for all of the training modules. However, various practical software engineering issues limit the availability of fault data for all modules in the training data. We present a semi-supervised learning scheme as a solution to software defect modeling when there is limited prior knowledge of software quality. The commonly used EM algorithm for estimating missing data values is used in conjunction with k-means clustering. An empirical investigation using software measurement and defect data from real world projects demonstrates the effectiveness and viability of the proposed method. It is shown that estimation accuracy of the defect prediction model after the semi-supervised learning process is generally better compared to a defect prediction model trained with a dataset consisting of (only available) program modules with known number of faults.

Keywords: Defects, Semi-supervised, Metrics, Clustering, EM algorithm

1. Introduction

The problem of software quality assurance is as vital as delivering a software product within the allocated budget and time constraints [1]. An important step toward equipping software engineers with a skill set that can help produce high-quality software is determining the body of knowledge needed to measure quality throughout the software development life cycle [2]. The key to developing high-quality software is the measurement and modeling of software quality. Various software quality improvement initiatives are employed in software engineering practice including reviews of software design and code, verification and validation, automated test case generation for additional testing, and re-engineering of low quality program modules.

*Corresponding author’s email: taghi@cse.fau.edu
Software quality models are very useful tools toward achieving the objectives of a software quality assurance initiative. Such models generally exploit the software engineering hypothesis that software measurements encapsulate the underlying quality of the software system. This assumption has been verified in numerous studies [3]. A software quality model is typically built or trained using software measurement and defect data of a similar, previously developed system. The trained model is then applied to the currently underdevelopment (also referred to as target system) system to estimate the quality or presence of defects in its program modules. Subsequently, the limited resources allocated for software quality inspection and improvement can be targeted toward only those program modules, achieving cost-effective resource utilization [4].

The above approach to software quality modeling makes two very important assumptions: (1) the development organization has experience with developing systems similar to the target project, and (2) software defect data is available for all program modules in the training data, i.e., software data of previously developed systems. We focus on problems associated with the latter assumption. Practical issues during software development often limit the availability of faults data associated with all instances (program modules) in the training data, i.e., only a small fraction of training modules are complete or have known number of faults. Consequently, the usual supervised learning approach to software faults modeling is inappropriate as the set of complete training data modules may not be large enough or sufficient to capture the underlying software quality trends of the system. Some issues contributing to this problem include cost of running data collection tools and inconsistency of data collection processes used (especially in a distributed development environment) among project components.

Toward addressing this problem, one approach would be learning from both the small set of complete training modules and the large set of training modules with unknown faults data, i.e., semi-supervised learning [5]. Such a strategy assumes that knowledge stored in the software measurements of the incomplete training modules can be used to improve software faults modeling. We present a semi-supervised solution to software faults modeling with limited prior knowledge on defect occurrence. The proposed approach uses the popular EM (Expectation-Maximization [6]) algorithm for estimating the unknown faults data of incomplete training modules and employs the popular \textit{k-means} clustering to facilitate augmentation of the training dataset with program modules without known defect data. To our knowledge there are no research works in the literature that investigate a semi-supervised learning scheme for software faults modeling and estimation.

Software measurement data obtained from multiple NASA software projects (JM1, KC1, KC2, and KC3 available at http://mdp.ivv.nasa.gov/) are used in our empirical investigation of the proposed semi-supervised learning approach for software faults modeling and estimation. To simulate the limited defect occurrence knowledge problem, a small sample of instances (program modules) is randomly selected from the JM1 dataset to represent the relatively-small set of training modules with known faults data. The remaining instances in the JM1 dataset are treated as the relatively-large set of training modules with unknown faults data. The performance of the semi-supervised learning scheme is then evaluated by using the KC1, KC2, and KC3 software projects as three independent test datasets. The initial number of JM1
modules with known faults is varied, and for each initial sample size we repeat the semi-supervised modeling process three times.

It is shown that software faults prediction (using a nearest-neighbor approach) for a given test dataset improves after the semi-supervised learning approach as compared to the predictions based on the initial small set of complete (i.e., known faults data) training program modules [7]. Hence, knowledge stored in software measurements of the training modules with unknown faults data is exploited to improve the software quality estimation model. Furthermore, the variation in size of the initial sample of modules with known faults indicated a prediction accuracy improvement with a larger initial sample size.

The remainder of the paper continues in the next section with a brief summary of some related works. Section 3 provides a concise description of the EM algorithm. Section 4 presents the proposed modeling approach. Section 5 details the empirical case study, including software measurement datasets and a detailed discussion on the obtained results. Finally, we conclude the paper in Section 6 and provide some suggestions for future work.

2. Related Work

A software quality model allows the software development team to track and detect potential software defects relatively early-on during development which is critical to many high-assurance systems. The practical application of software quality models at different phases in the development life-cycle depends on availability and types of software measurements.

In the literature, software measurements and software fault (defect) data have been used in the development of models that predict software quality. A software quality classification model [8, 9, 10] predicts the fault-proneness membership of program modules while a software fault prediction model [11, 12, 13] estimates the number of defects expected in a program module. Schneidewind [14] utilizes logistic regression in combination with Boolean discriminant functions for predicting fault-prone program modules. Guo et al. [15] predict fault-prone program modules using Dempster-Shafer networks. Some other works on software quality modeling and estimation include Imam et al. [16] and Pizzi et al. [17]. Compared to those works, our study provides a unique perspective into the problem of software quality modeling when there is limited availability of defect data from past projects.

In a machine learning problem when both labeled and unlabeled data are used during the learning process, it is termed as semi-supervised learning [5, 18]. Considering the software quality modeling and estimation problem, the labeled dataset would consist of program modules with known software measurements and known quality/defect data; and the unlabeled dataset would consist of program modules with known software measurements but unknown quality/defect data. In a semi-supervised learning scheme the labeled dataset is iteratively augmented with instances (with predicted dependent variable, such as class label) from the unlabeled dataset based on some selection measure [19]. While almost all semi-supervised learning related work focuses on the classification problem, we investigated the semi-supervised problem for a quantitative prediction model. However, we summarize some important work related to semi-supervised learning.

Semi-supervised classification schemes have been investigated across various domains, including content-based image retrieval [20], human motion and gesture pattern recognition...
Some of the recently investigated techniques for semi-supervised classification include the EM algorithm [22], co-training [18, 23], active learning [24], and support vector machine (SVM) [25, 26].

The EM algorithm and its variations have recently been used for semi-supervised classification of text documents [27, 28]. Nigam et al. [22] investigate a semi-supervised scheme for text classification based on a combination of the EM algorithm and a Naive Bayes classifier. The approach initially trains a classifier with the available labeled documents, and then uses the classifier to assign probabilistically weighted class labels to the unlabeled document by computing the expectation of the missing class labels. They combine EM with Naive Bayes since, as a mixture of multinomials, it is commonly used for modeling text classification.

The co-training approach involves two different supervised learning algorithms that are used to complement each other based on the assumption that each algorithm would learn a different hypothesis from the labeled dataset [18, 29, 30]. A basic assumption with co-training is that the instances can be adequately classified by two redundant sets of sufficient features [29]. Fung et al. [26] present a concave minimization approach for classifying unlabeled data based on a combination of a clustering technique and a linear SVM. The \( k \)-median clustering algorithm is used to select representative data points which are then labeled by an expert. The concave minimization approach is then applied with the linear SVM to classify the unlabeled instances.

3. **EM Algorithm for Missing Data**

The Expectation Maximization (EM) algorithm is a general iterative method for maximum likelihood estimation in data mining problems with incomplete data [6]. The essence of EM is a formalization of a relatively intuitive procedure for handling missing data. The EM algorithm takes an iterative approach consisting of replacing missing data with estimated values, estimating model parameters, and re-estimating the missing data values. Each iteration of EM consists of an E or Expectation step and an M or Maximization step, with each having a direct statistical interpretation.

In our study, the number of faults in program modules with unknown defect data is treated as missing data, and EM is then used to estimate the missing values. The EM algorithm implemented for our study on semi-supervised software faults modeling is based on maximum likelihood estimation of missing data, means, and covariances for multivariate normal samples [6]. Consider that \((Y_1, Y_2, \ldots, Y_K)\) have a \(K\)-variate normal distribution with mean \(\mu = (\mu_1, \mu_2, \ldots, \mu_K)\) and covariance matrix \(\Sigma = (\sigma_{jk})\) where \(j, k = 1, \ldots, K\). If \(Y\) represents a random sample of size \(n\) on \((Y_1, Y_2, \ldots, Y_K)\), \(Y_{obs}\) represents the set of observed (non-missing) values, and \(Y_{miss}\) represents the missing data, then we summarize \(Y = (Y_{obs}, Y_{miss})\). In addition, let \(Y_{obs} = (y_{obs,1}, y_{obs,2}, \ldots, y_{obs,n})\) where \(y_{obs,i}\) is the set
of variables observed for case \(i, i = 1, 2, \ldots, n\). The loglikelihood based on the observed data is then given by,

\[
\ell(\mu, \sum | Y_{obs}) = C - \frac{1}{2} \sum_{i=1}^{n} \ln|\sum_{obs,i}| - \frac{1}{2} \sum_{i=1}^{n} (y_{obs,i} - \mu_{obs,i})^T \sum_{obs,i}^{-1} (y_{obs,i} - \mu_{obs,i}) \tag{1}
\]

where \(C\) is a constant, and \(\mu_{obs,i}\) and \(\sum_{obs,i}\) are the mean and covariance (Cov) matrices of the non-missing components of \(Y\) for observation \(i\). In order to obtain the EM algorithm for maximizing Equation (1), it should be noted that the hypothetical complete data \(Y\) belong to the regular exponential family with sufficient statistics \(S\) [6] and is given by,

\[
S = \left( \sum_{i=1}^{n} y_{ij} ; \sum_{i=1}^{n} y_{ij} y_{ik} \right)
\]

where \(j, k = 1, \ldots, K\). At the \(t^{th}\) EM iteration, \(\theta^{(t)} = (\mu^{(t)}, \sum^{(t)})\) denotes the current estimates of the parameters. The E step of the algorithm involves computing,

\[
E\left( \sum_{i=1}^{n} y_{ij} | Y_{obs}, \theta^{(t)} \right) = \sum_{i=1}^{n} y_{ij}^{(t)}
\]

and

\[
E\left( \sum_{i=1}^{n} y_{ij} y_{ik} | Y_{obs}, \theta^{(t)} \right) = \sum_{i=1}^{n} \left( y_{ij}^{(t)} y_{ik}^{(t)} + c_{jki}^{(t)} \right)
\]

where \(y_{ij}^{(t)} = y_{ij}\) if \(y_{ij}\) is observed, or \(y_{ij}^{(t)} = E(y_{ij} | y_{obs}, \theta^{(t)})\) if \(y_{ij}\) is missing; and \(c_{jki}^{(t)} = 0\) if \(y_{ij}\) or \(y_{ik}\) are observed, or \(c_{jki}^{(t)} = Cov(y_{ij}, y_{ik} | y_{obs}, \theta^{(t)})\) if \(y_{ij}\) or \(y_{ik}\) are missing.

The missing values \(y_{ij}\) are thus substituted by the conditional mean of \(y_{ij}\) given the set of non-missing values \(y_{obs,i}\) observed for that case. The conditional means and the non-zero conditional covariances are found from the current parameter estimates by sweeping the augmented covariance matrix such that the variables \(y_{obs,i}\) are predictors in the regression equation and the remaining variables are output variables. The sweeping operator is described in [6]. The M step of the EM algorithm involves computing the new estimates \(\theta^{(t+1)}\) of the parameters from the estimated complete-data sufficient statistics. More specifically, \(\mu_{j}^{(t+1)}\) and \(\sigma_{jk}^{(t+1)}\) are respectively computed as follows,

\[
\mu_{j}^{(t+1)} = \frac{1}{n} \sum_{i=1}^{n} y_{ij}^{(t)}
\]

and
The iterative E step and M step of the EM algorithm continue until a stopping criterion is reached. In our study, the EM iteration is stopped if the maximum change among the means or covariances between two consecutive iterations is less than 0.0001. The initial values of the parameter set are obtained by estimating means and variances from all available values of each variable, and then estimating covariances from all available pairwise values using the computed means.

4. Semi-Supervised Software Faults Modeling Approach

Given the relatively-small set of training program modules with known defect data, FA, and the relatively-large set of training program modules with unknown defect data, FU, the following summarizes our proposed semi-supervised learning for software faults modeling with limited defect occurrence knowledge.

1. Group FA into $CFA_1$, $CFA_2$, ..., $CFA_p$ clusters using $k$-means. Clustering includes the faults variable.
2. Given FA and FU, estimate the unknown (missing) faults data of FU by applying the EM algorithm described in Section 3.
3. Group FU into $CFU_1$, $CFU_2$, ..., $CFU_q$ clusters using $k$-means. Clustering includes the EM-estimated faults variable.
4. For a given significance level $\alpha$ conduct $t$-test (on the faults variable) between each $CFU_i$, where $1 \leq i \leq q$, and all $CFA_j$, where $1 \leq j \leq p$, to determine whether the two respective data samples ($CFU_i$ and $CFA_j$) are from the same normal distribution. We use $\alpha = 0.10$ as the significance level.
5. For a given cluster $CFU_i$, add its instances to FA if its $t$-test is not significant for any $CFA_j$, where $1 \leq j \leq p$. The instances of $CFU_i$, where $1 \leq i \leq q$, are added with their EM-estimated number of faults values.
6. The updated FA and FU training datasets are obtained.

The software faults prediction for multiple independent test datasets is obtained by applying a nearest-neighbor prediction strategy. The number of faults for a test data program module is estimated as the mean number of faults of its $nn$ nearest neighbors in the training dataset. The test dataset prediction is then evaluated by computing the average absolute error ($aae$) and average relative error ($are$) [13].

We compare the software faults prediction accuracy of the nearest-neighbor model built before the semi-supervised learning process with the prediction accuracy of a similar nearest-neighbor model built after the semi-supervised learning process. The training data before semi-supervised learning is the initial FA dataset selected from JM1. The training dataset after semi-supervised learning consists of both the initial FA and program modules augmented (with predicted number of faults) from FU during our semi-supervised learning. A nearest-
neighbor prediction model is built using both sets of training data and then applied to the respective test datasets.

5. Empirical Case Study

5.1. Software Measurement Data

The software measurements and quality data used in our study to investigate the proposed semi-supervised clustering approach is that of a large NASA software project, JM1. Written in C, the JM1 project is a real-time ground system that uses simulations to generate certain predictions for missions. The data was made available through the Metrics Data Program at NASA (http://mdp.ivv.nasa.gov/), and included software measurement data and associated error data collected at the function level. Hence, a program module for the system consisted of a function or method.

The fault data collected for the system represents, for a given module, faults detected during software development. The original JM1 dataset consisted of 10,883 software modules, of which 2,105 modules had software defects (ranging from 1 to 26) while the remaining 8,778 modules were defect-free, i.e., had no software faults. We note that the terms errors, defects, and faults are used interchangeably in this paper.

<table>
<thead>
<tr>
<th>Table 1: Software Measurements</th>
</tr>
</thead>
<tbody>
<tr>
<td>Category</td>
</tr>
<tr>
<td>----------</td>
</tr>
<tr>
<td>Line Count</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Halstead</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>McCabe</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Branch</td>
</tr>
</tbody>
</table>

The original JM1 dataset consisted of 10,883 software modules, of which 2,105 modules had errors (ranging from 1 to 26) while the remaining 8,778 modules were error-free, i.e., had no software faults. In the case of JM1, upon removing inconsistent modules [31] the dataset was reduced from 10,883 to 8,850 modules. This dataset consisted of 1,687 modules with 1 or more defects and 7,163 modules with zero defects.

Each program module in the JM1 was characterized by 21 software measurements [3, 15]: 13 metrics as shown in Table 1, and eight derived Halstead metrics, i.e., Halstead Length, Halstead Volume, Halstead Level, Halstead Difficulty, Halstead Content, Halstead Effort,
Halstead Error Estimate, and Halstead Program Time. We only used the 13 basic software metrics in our analysis. The eight derived Halstead metrics were not used.

The metrics for the JM1 (and other software measurement datasets) were primarily governed by their availability, internal workings of the projects, and the data collection tools used. The type and numbers of metrics made available were determined by the NASA Metrics Data Program. Other metrics, including software process measurements, were not available. The use of the specific software metrics does not advocate their effectiveness, and a different project may consider a different set of software measurements for analysis [3, 16].

In our semi-supervised modeling approach, the JM1 dataset is used as a training data for software quality analysis. In order to gauge the performance of the semi-supervised modeling results, we use software measurement data of three other NASA projects, KC1, KC2, and KC3 as test datasets. These software measurement datasets were also obtained through the NASA Metrics Data Program. A program module of these projects also consisted of a function, subroutine, or method.

The three test data projects were characterized by the same software product metrics used for the JM1 project, and were built in a similar software development organization. Evaluating a given software quality model with an independent test dataset simulates the application on a similar currently under-development software project. The test datasets are described below:

- The KC1 project is a single CSCI within a large ground system and consists of 43 KLOC (thousand lines of code) of C++ code. A given CSCI comprises of logical groups of computer software components (CSCs). The dataset contains 2107 modules, of which 325 have one or more faults and 1782 have zero faults. The maximum number of faults in a module is seven.

- The KC2 project, written in C++, is the science data processing unit of a storage management system used for receiving and processing ground data for missions. The original KC2 dataset consisted of over 3000 software modules; however, for our study only 520 modules were considered because they were developed by NASA software developers and were not COTS software. The dataset contains 520 modules, of which 106 have one or more faults and 414 have zero faults. The maximum number of faults in a software module is 13.

- The KC3 project, written in 18 KLOC of Java, is a software application that collects, processes, and delivers satellite meta-data. The dataset contains 458 modules, of which 43 have one or more faults and 415 have zero faults. The maximum number of faults in a module is six.

5.2. Empirical Settings

The initial \( FA \) dataset was obtained by randomly selecting a given percentage of the 8850 JM1 program modules [7]. The selection strategy maintains the approximate 80:20 proportions of program modules with zero faults and modules with one or more faults in the JM1 dataset. Initially, we studied the proposed approach with \( FA = 885 \), i.e., 10% of the JM1 modules. For that study, the initial \( FA \) dataset consisted of 708 modules with 0 faults and 177 modules with
one or more faults. The remaining JM1 modules constituted the $FU$ dataset. Empirical studies were conducted with three such samples of JM1 and the averages of their respective results are reported.

In order to explore the effect of the initial $FA$ dataset size, we studied two more sets of experiments with different $FA$ values. We considered 5% (443, i.e., 354 + 89) and 20% (1770, i.e., 1416 + 354) of the 8850 JM1 program modules. Similar to our analysis with 10% of the JM1 modules, these two sets of empirical studies were performed by randomly selecting three samples for a given $FA$ value and reporting the average results over the respective samples. We considered these three $FA$ values to represent very small, small, and medium sizes for the dataset consisting of program modules with known number of faults.

The empirical settings used during the $k$-means clustering include: 20 clusters for the initial $FA$ dataset, 50 clusters for the initial $FU$ dataset; 200 iterations for each restart or replication; 20 restart or replication; random instance selection for initial cluster centroids; and Euclidean distance function for clustering. These settings were chosen based on similar options selected in our prior studies on semi-supervised software quality modeling [31]. A more comprehensive study would shed light on whether optimizing these different empirical parameters makes any significant impact on the proposed method’s performance. This is planned as one of our future research.

5.3. Results and Discussion

We first present detailed empirical results obtained with the KC2 test dataset [7]. Corresponding results obtained with the KC1 and KC3 test datasets are summarized to avoid duplication of empirical observations. The software faults prediction error results for the KC2 dataset for the three different initial $FA$ values are shown in Tables 2, 3 and 4. For each of the three $FA$ values, the corresponding table shows results based on the initial $FA$ dataset, i.e., before semi-supervised learning, and results based on the final $FA$ dataset, i.e., after semi-supervised learning. For example, Table 2 represents prediction errors for the initial and final models with $FA = 443$, while Table 4 provides the corresponding results for $FA = 1770$.

The tables show the average absolute ($aae$) and average relative ($are$) errors computed for different values of the nearest-neighbor parameter, $nn$. The KC2 dataset results indicate that generally both $aae$ and $are$ improve (get lower) with an increase in the $FA$ dataset size. The improvement is more pronounced when moving from $FA = 443$ to $FA = 885$, compared to when moving from $FA = 885$ to $FA = 1770$. This may reflect the fact that relatively little knowledge is gained from the software measurements of the $FU$ dataset when the initial $FA$ is not very small.

The nearest-neighbor value is varied from 3 to 35 with intermittent values as shown in the KC2 tables. Instead of optimizing the best prediction value for $nn$, we opted to demonstrate the effectiveness of the semi-supervised learning process by showing the prediction error rates over multiple $nn$ values. A true software fault prediction model, however, can potentially be determined by optimizing $nn$ over the given $FA$ dataset and then using the optimized $nn$ value for test dataset prediction. A very minimal variation in $aae$ and $are$ values is noticed across the multiple $nn$ values. This may be a characteristic of the JM1 software measurement dataset.
Table 2: KC2 Error Rates with $FA = 443$

<table>
<thead>
<tr>
<th>$nn$</th>
<th>$aae$</th>
<th>$are$</th>
<th>$nn$</th>
<th>$aae$</th>
<th>$are$</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>0.3348</td>
<td>0.2574</td>
<td>3</td>
<td>0.3003</td>
<td>0.2158</td>
</tr>
<tr>
<td>5</td>
<td>0.3397</td>
<td>0.2632</td>
<td>5</td>
<td>0.3047</td>
<td>0.2205</td>
</tr>
<tr>
<td>10</td>
<td>0.3314</td>
<td>0.2486</td>
<td>10</td>
<td>0.3024</td>
<td>0.2176</td>
</tr>
<tr>
<td>15</td>
<td>0.3351</td>
<td>0.2593</td>
<td>15</td>
<td>0.3018</td>
<td>0.2152</td>
</tr>
<tr>
<td>25</td>
<td>0.3356</td>
<td>0.2555</td>
<td>25</td>
<td>0.3015</td>
<td>0.2163</td>
</tr>
<tr>
<td>35</td>
<td>0.3360</td>
<td>0.2549</td>
<td>35</td>
<td>0.3009</td>
<td>0.2141</td>
</tr>
</tbody>
</table>

Table 3: KC2 Error Rates with $FA = 885$

<table>
<thead>
<tr>
<th>$nn$</th>
<th>$aae$</th>
<th>$are$</th>
<th>$nn$</th>
<th>$aae$</th>
<th>$are$</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>0.2218</td>
<td>0.1469</td>
<td>3</td>
<td>0.1942</td>
<td>0.1186</td>
</tr>
<tr>
<td>5</td>
<td>0.2257</td>
<td>0.1504</td>
<td>5</td>
<td>0.1982</td>
<td>0.1225</td>
</tr>
<tr>
<td>10</td>
<td>0.2205</td>
<td>0.1452</td>
<td>10</td>
<td>0.1972</td>
<td>0.1211</td>
</tr>
<tr>
<td>15</td>
<td>0.2239</td>
<td>0.1476</td>
<td>15</td>
<td>0.1969</td>
<td>0.1204</td>
</tr>
<tr>
<td>25</td>
<td>0.2224</td>
<td>0.1451</td>
<td>25</td>
<td>0.1960</td>
<td>0.1194</td>
</tr>
<tr>
<td>35</td>
<td>0.2226</td>
<td>0.1443</td>
<td>35</td>
<td>0.1945</td>
<td>0.1176</td>
</tr>
</tbody>
</table>

Table 4: KC2 Error Rates with $FA = 1770$

<table>
<thead>
<tr>
<th>$nn$</th>
<th>$aae$</th>
<th>$are$</th>
<th>$nn$</th>
<th>$aae$</th>
<th>$are$</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>0.1721</td>
<td>0.1132</td>
<td>3</td>
<td>0.1314</td>
<td>0.0976</td>
</tr>
<tr>
<td>5</td>
<td>0.1797</td>
<td>0.1244</td>
<td>5</td>
<td>0.1362</td>
<td>0.1158</td>
</tr>
<tr>
<td>10</td>
<td>0.1775</td>
<td>0.1207</td>
<td>10</td>
<td>0.1344</td>
<td>0.1125</td>
</tr>
<tr>
<td>15</td>
<td>0.1743</td>
<td>0.1201</td>
<td>15</td>
<td>0.1340</td>
<td>0.1119</td>
</tr>
<tr>
<td>25</td>
<td>0.1736</td>
<td>0.1160</td>
<td>25</td>
<td>0.1328</td>
<td>0.1107</td>
</tr>
<tr>
<td>35</td>
<td>0.1729</td>
<td>0.1118</td>
<td>35</td>
<td>0.1319</td>
<td>0.0943</td>
</tr>
</tbody>
</table>

Given the very minimal variation in the $aae$ and $are$ prediction error values with respect to $nn$, empirical results for the KC1 and KC3 test datasets are presented as an average over the different $nn$ values. We determined that no information is lost in doing so, and the empirical observations and deductions remained the same. The software fault prediction error performances for the KC1 and KC2 test datasets are summarized in Table 5, where prediction error rates for each of the three $FA$ values are shown.

Compared to KC2, the prediction errors for KC3 are relatively higher. This may reflect the characteristic of KC3 that it has an approximately 90:10 ratio of number of modules with 0 faults to number of modules with 1 or more faults. The marked difference of KC3 from the corresponding 80:20 ratio of the training dataset, i.e., JM1, may have led to the poor
prediction performance for the KC3 program modules. However, the relative improvement between the final and initial models of KC3 still follows the trend observed with KC2. The prediction error results for the third test dataset, i.e., KC1, also confirms the observation that our semi-supervised learning process improves fault prediction compared to a model built on the small training (available) dataset of program modules with known faults data.

<p>| Table 5: KC1 and KC3 Average Error Rates |
|----------------------------------------|-----------------|-----------------|-----------------|-----------------|</p>
<table>
<thead>
<tr>
<th>FA</th>
<th>Stage</th>
<th>aae</th>
<th>are</th>
<th>aae</th>
<th>are</th>
</tr>
</thead>
<tbody>
<tr>
<td>443</td>
<td>Initial</td>
<td>0.2538</td>
<td>0.1958</td>
<td>0.4135</td>
<td>0.3536</td>
</tr>
<tr>
<td></td>
<td>Final</td>
<td>0.2102</td>
<td>0.1369</td>
<td>0.3701</td>
<td>0.2820</td>
</tr>
<tr>
<td>885</td>
<td>Initial</td>
<td>0.1935</td>
<td>0.1257</td>
<td>0.3593</td>
<td>0.2703</td>
</tr>
<tr>
<td></td>
<td>Final</td>
<td>0.1462</td>
<td>0.0739</td>
<td>0.3148</td>
<td>0.2274</td>
</tr>
<tr>
<td>1770</td>
<td>Initial</td>
<td>0.1583</td>
<td>0.1205</td>
<td>0.2636</td>
<td>0.2069</td>
</tr>
<tr>
<td></td>
<td>Final</td>
<td>0.1176</td>
<td>0.1061</td>
<td>0.1958</td>
<td>0.1537</td>
</tr>
</tbody>
</table>

The most practical and important observation in the tables presented in this section is the reduction in aae and are errors between the respective final and initial prediction models. This was true for all \( nn \) values considered in our study. The reduction in the test datasets prediction errors acknowledges and confirms the software quality modeling benefit gained via the proposed semi-supervised learning process.

6. Conclusion

Software quality and reliability engineering is an important component of software project development. While different techniques and processes can be used toward improving software quality, a knowledge-based approach to software quality modeling and estimation provides practical guidance in improving the cost-benefit of software project resources. Software quality models are very useful tools toward achieving the objectives of a software quality assurance initiative.

In related literature a software quality model is typically built using software measurement and defect data of a similar, previously developed, system. It is then used to estimate the quality or presence of defects in program modules of the target system. Such a supervised learning approach to software quality modeling makes two very important assumptions: (1) the software development organization has experience with developing systems similar to the target project, and (2) software defect data is available for all program modules in the training data, i.e., software data of similar, previously developed, systems.

However, if the software defect data is not available for all program modules of the past project(s), then a supervised learning strategy may be inappropriate. Given a software measurement dataset consisting of a relatively small fraction of program modules with known software faults data and a relatively large number of modules with unknown faults data, a semi-supervised learning approach to knowledge-based software quality modeling makes more sense.
This paper presented a semi-supervised learning process aimed at addressing the problem of software defect prediction when there is limited availability of software faults data from past experiences. The proposed approach is shown to yield promising results based on observations made in our empirical case study of multiple real-world software projects. The proposed semi-supervised learning approach uses the popular EM algorithm for estimating missing data values, which in our case are the unknown faults data of relatively large number of (incomplete) training program modules. In addition, the approach employs k-means clustering to facilitate augmentation of the training dataset with program modules without known defect data.

The reduction in the software faults prediction error rates from the initial to post-semi-supervised learning stages provides good indication of the practical benefit of the proposed approach. Additional empirical case studies with different empirical settings may provide further validation. Some other studies related to this paper can: investigate possible reasons why certain program modules were not selected to augment the FA dataset; consider clustering algorithms other than k-means and investigate other distance measures during clustering; and, consider applying the semi-supervised learning scheme with many different proportions of the initial FA dataset size.

References


**Naeem Seliya** is an Assistant Professor of the Computer and Information Science Department at the University of Michigan – Dearborn. He received his Ph.D. in Computer Engineering from Florida Atlantic University, Boca Raton, FL, USA in 2005. His research interests include software engineering, data mining and machine learning, application and data security, computational intelligence, and medical informatics. He has over 30 publications in refereed technical journals, conferences and symposia. He is a member of IEEE and ACM.

**Taghi M. Khoshgoftaar** is a Professor of the Computer Science and Engineering Department, Florida Atlantic University, and the Director of the Empirical Software Engineering Laboratory, and the 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 350 refereed papers in these areas. He is a member of the IEEE, IEEE Computer Society, and 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. He is the Program Chair of the 20th International Conference on Software Engineering and Knowledge Engineering in 2008. He has served on technical program committees of various international conferences, symposia, and workshops. Also, he has served as North American Editor of the Software Quality Journal, and is on the editorial boards of the journals Software Quality and Fuzzy Systems.