Available online at www.sciencedirect.com
ScienceDirect
The Journal of Systems and Software 81 (2008) 186–195
The Journal of Systems and Software
www.elsevier.com/locate/jss

Applying machine learning to software fault-proneness prediction

Iker Gondra *

Department of Mathematics, Statistics, and Computer Science, St. Francis Xavier University, P.O. Box 5000, Antigonish, Canada NS B2G 2W5

Available online 7 June 2007

Abstract

The importance of software testing to quality assurance cannot be overemphasized. The estimation of a module's fault-proneness is important for minimizing cost and improving the effectiveness of the software testing process. Unfortunately, no general technique for estimating software fault-proneness is available. The observed correlation between some software metrics and fault-proneness has resulted in a variety of predictive models based on multiple metrics. Much work has concentrated on how to select the software metrics that are most likely to indicate fault-proneness. In this paper, we propose the use of machine learning for this purpose. Specifically, given historical data on software metric values and number of reported errors, an Artificial Neural Network (ANN) is trained. Then, in order to determine the importance of each software metric in predicting fault-proneness, a sensitivity analysis is performed on the trained ANN. The software metrics that are deemed to be the most critical are then used as the basis of an ANN-based predictive model of a continuous measure of fault-proneness. We also view fault-proneness prediction as a binary classification task (i.e., a module can either contain errors or be error-free) and use Support Vector Machines (SVM) as a state-of-the-art classification method. We perform a comparative experimental study of the effectiveness of ANNs and SVMs on a data set obtained from NASA's Metrics Data Program data repository.

Keywords: Software testing; Software metrics; Fault-proneness; Machine learning; Neural network; Sensitivity analysis; Support vector machine

1. Introduction

There has been a tremendous growth in the demand for software quality during recent years. As a consequence, issues related to testing are becoming increasingly crucial. The ability to measure software fault-proneness can be extremely important for minimizing cost and improving the overall effectiveness of the testing process. The majority of faults in a software system are found in a few of its components (Boehm and Papaccio, 1988; Porter and Selby, 1990). The "80:20" rule states that approximately 20% of a software system is responsible for 80 percent of its errors, costs, and rework (Boehm, 1987). Boehm and Papaccio (1988) conclude:

> "The major implication of this distribution is that software verification and validation activities should focus on identifying and eliminating the specific high-risk problems to be encountered by a software project, rather than spreading their available early-problem-elimination effort uniformly across trivial and severe problems."

Therefore, the possibility of estimating the potential faultiness of software components at an early stage can be of great help for planning testing activities. For example, testing effort can be focused on a subset of software components (e.g., the potentially "most troublesome" 20%). Unfortunately, software fault-proneness cannot be directly measured. However, it can be estimated based on software metrics, which provide quantitative descriptions of program attributes. A number of studies provide empirical evidence that correlation exists between some software metrics and fault-proneness (see, for example, Gill and Kemerer, 1991; Khoshgoftaar and Allen, 1998; Lehman et al., 1998; Li and Cheung, 1987; Shen et al., 1985). Early research on software metrics has focused mostly on Lines of Code (LOC), McCabe (1976), and Halstead (1977) metrics.

* Tel.: +1 902 867 3314; fax: +1 902 867 3302.
E-mail address: igondra@stfx.ca

0164-1212/$ - see front matter © 2007 Elsevier Inc. All rights reserved.
doi:10.1016/j.jss.2007.05.035

Several methods have been explored to develop predictive models of software fault-proneness. Statistical techniques (e.g., Discriminant Analysis Briand et al., 1993; Munson and Khoshgoftaar, 1992, Factor Analysis Khoshgoftaar and Munson, 1990; Munson and Khoshgoftaar, 1992; Munson and Khoshgoftaar, 1990), machine learning techniques (e.g., Decision Trees Porter and Selby, 1990, Artificial Neural Networks (ANN) Khoshgoftaar et al., 1994; Neumann, 2002, Support Vector Machines (SVM) Xing et al., 2005), and many others (Myrtveit et al., 2005) have been proposed. Much work has concentrated on how to select the software metrics that are more likely to indicate fault-proneness. For example, Principal Component Analysis (PCA) (Jolliffe, 1986) is used in both Neumann (2002) and Xing et al. (2005) to reduce the number of software metrics while retaining most of the observed variation. It can be proven that the representation given by PCA is an optimal linear reduction in the mean-square sense (Jolliffe, 1986). The basic procedure consists of finding a number of orthonormal vectors (i.e., eigenvectors) that form a basis for the data. Those vectors are the "principal components" and the data are linear combinations of them. It turns out that the projected data shows the most variance on the first principal component, the next highest variance on the second principal component, and so on. Thus, the dimensionality of the data can be reduced simply by eliminating the last principal components (i.e., the ones with smallest eigenvalues that do not account for much of the variance in the data). A similar technique, Factor Analysis, is used in Khoshgoftaar and Munson (1990), Munson and Khoshgoftaar (1992) and Munson and Khoshgoftaar (1990). A disadvantage of those approaches is that they generate metrics that do not have an intuitive and easy interpretation in terms of program characteristics. Although it is clear that such techniques have an important role, it would be difficult to give advice to a programmer or designer on how to modify the program or design so as to optimize the value of a principal component metric.

The relationships between software metrics and fault-proneness are often complex. Thus, the adequacy of traditional linear models is compromised. This has resulted in the development of non-linear models (e.g., ANN) which are expected to provide superior performance than their linear counterparts. In this paper, we propose the use of machine learning for selecting a subset of software metrics that are most likely to predict the existence of errors. Specifically, given historical data on software metric values and number of reported errors, an ANN is trained. Then, in order to determine the importance of each software metric in predicting fault-proneness, a sensitivity analysis is performed on the trained ANN. The software metrics that are deemed to be the most critical are then used as the basis of fault-proneness prediction models based on machine learning. We use ANNs to predict the value of a continuous measure of fault-proneness. The fault-proneness prediction problem can also be regarded as a classification task (i.e., a module is predicted to either contain errors or be error-free). This motivated us to use SVMs as a state-of-the-art classification method. We perform a comparative experimental study of the effectiveness of ANNs and SVMs on a data set obtained from NASA's Metrics Data Program data repository.[1] The application of SVMs to the fault-proneness prediction problem has been proposed only very recently in Xing et al. (2005). To the best of our knowledge, there is no previous work that compares the performance of these two important machine learning approaches on the fault-proneness prediction problem.

The remainder of this paper is organized as follows. In the next section we summarize basic concepts of machine learning, ANNs, SVMs, and sensitivity analysis. Section 3 presents our methodology. A description of the data used and experimental results are presented in Section 4. The paper ends with some conclusions and ideas for future work in this area.

2. Machine learning

The field of machine learning focuses on the study of algorithms that improve their performance at some task automatically through experience (Mitchell, 1997). Suppose we are given training data as a set of n observations. Each observation is a pair $(\mathbf{x}_i, y_i)$ where $\mathbf{x}_i \in \Re^d$ and $y_i \in \Re$. Assume that the training data has been drawn independently from some unknown cumulative probability distribution $P(\mathbf{x}, y)$. The goal is to find a function (or model) $f : \Re^d \mapsto \Re$ that implements the optimal mapping. In order to make learning feasible, we have to specify a function space $\mathscr{F}$ from which the function is to be chosen.

In particular, given training data $\{(\mathbf{x}_i, y_i)\}_1^n$ for a binary classification task where $\mathbf{x}_i \in \Re^d$ and $y_i \in \{1, -1\}$ is the class label. Assume that the data is linearly separable and let $\mathscr{F}$ be the set of linear decision boundaries of the form

$$f(\mathbf{x}) = \text{sign}(\mathbf{w} \cdot \mathbf{x} + b) \tag{1}$$

where $\mathbf{w} \in \Re^d$ and $b \in \Re$ are the adjustable parameters. Thus, choosing particular values for $\mathbf{w}$ and b results in a trained classifier. For any trained classifier, the hyperplane corresponding to $\mathbf{w} \cdot \mathbf{x} + b = 0$ is the decision boundary (see Fig. 1). A linear decision boundary is a simple classifier that can be learned very efficiently. However, it can correctly classify data that is linearly separable only. On the other hand, a more complex decision boundary can correctly classify general data that may not be linearly separable.

In general, for any $\mathscr{F}$, one way to measure the performance of a trained function $f \in \mathscr{F}$ is to look at the mean error computed from the training data. This is known as the training error (or empirical risk). Minimizing the training error is one of the most commonly used optimization procedures. However, even when there is no error in the

[1] http://mdp.ivv.nasa.gov.

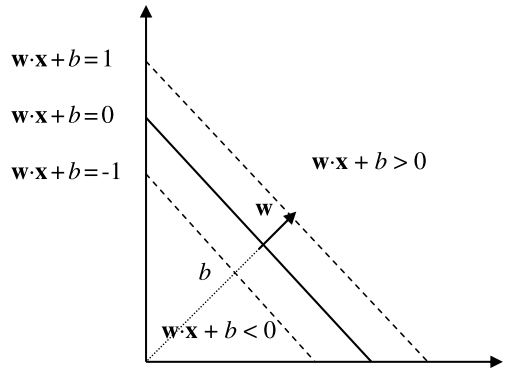

Fig. 1. A simple binary classifier. The hyperplane corresponding to $\mathbf{w}\cdot\mathbf{x}+b=0$ is the decision boundary.

training data, the function may not generate correct values for previously unseen data. This problem is known as overfitting. The ability of a function to generate correct values for new data that is not in the training set is known as generalization. Having a function with good generalization is, of course, a much harder problem. There is a competition of terms. As the complexity of the function increases, the training error tends to decrease. However, the generalization error usually increases with increasing complexity (see Fig. 2). Thus, for such models with broad approximation abilities and few specific assumptions (e.g., ANNs), the distinction between memorization and generalization becomes important (Gershenfeld and Weigend, 1993). Cross-validation is a technique that reduces overfitting. In simple holdout cross-validation, a validation set (disjoint from the training set) is used for model selection (i.e., to assess generalization performance and determine when to stop the training process so as to avoid overfitting). Notice that if true generalization error estimates are to be computed, an independent test set used only to asses the performance of the final model (i.e., after stopping the training process) has to be used. This is because the generalization performance estimate provided by the validation set is biased (i.e., the stopping point in the training process is optimal with respect to the validation set, but may not be optimal for an independent test set) and thus tends to underestimate the true generalization performance.

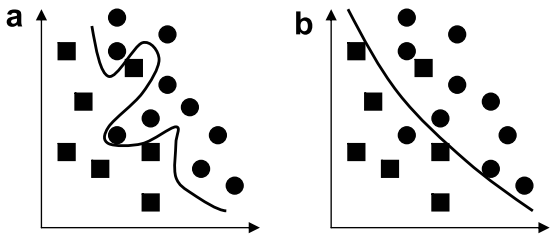

Fig. 2. Generalization performance: (a) an overly complex classifier that results in zero training error, but may not generalize well to unseen data; (b) a classifier that might represent the optimal tradeoff between training error and complexity, thus capable of generalizing well to unseen data.

2.1. Artificial Neural Networks

Artificial Neural Networks (ANN) are a class of non-linear, non-parametric models that can be trained to approximate general non-linear, multivariate functions. They are massively parallel systems comprised of many interconnected processing elements (known as nodes or neurons) based on neurobiological models of the brain. One of their main advantages in comparison to other models is that they have very few domain-specific assumptions and are highly adaptable (i.e., they learn from experience). Thus, they need no *a priori* assumption of a model and are capable of inferring complex non-linear input–output transformations. Therefore, they can be easily applied to problems whose solutions require knowledge which is difficult to specify but for which there is an abundance of examples. That is, in contrast to traditional models, which are "theory-rich" and "data-poor", ANNs are "data-rich" and "theory-poor" in a way that little or no *a priori* knowledge of the problem is present (Gershenfeld and Weigend, 1993). Besides their typical use in (non-linear) regression, they are commonly used in pattern recognition, where the ANN assigns a set of input features to one or more classes.

In the late 1950s, Rosenblatt (1958) introduced the perceptron learning rule (see Fig. 3), the first iterative algorithm for training a simple ANN: the perceptron (see Fig. 4), which is a linear classifier. After initializing $\mathbf{w}$ and b randomly, each training point $\mathbf{x}_i$ is presented and the output value y is compared against y_i. If y and y_i are different (i.e., $\mathbf{x}_i$ is misclassified) the values of $\mathbf{w}$ and b are adapted by moving them either towards or away from $\mathbf{x}_i$. Rosenblatt proved that, assuming the classes are linearly separable, the algorithm will always converge and find values for $\mathbf{w}$ and b that solve the classification problem. That is, the algorithm finds a hyperplane that divides the d-dimensional space into two decision regions (see Fig. 1).

The key idea responsible for the power, potential, and popularity of ANNs is the insertion of one or more layers of non-linear hidden units (between the input and output layers) (Gershenfeld and Weigend, 1993). These non-linearities allow for interactions between the inputs (e.g., products between input variables). As a result, the network can fit more complicated functions (Gershenfeld and Weigend, 1993). A multilayer ANN consists of an input

Given training set $\{(\mathbf{x_i}, y_i)\}_1^n$ and learning rate $\eta \in \Re$
Initialize $\mathbf{w}$ and b to small random values
  Repeat
    For i = 1 to n
      If $y_i f(\mathbf{x_i}) \leq 0$ (if misclassification)
        $\mathbf{w} \leftarrow \mathbf{w} + \eta y_i \mathbf{x_i}$
        $b \leftarrow b + \eta y_i$
    End For
  Until no misclassifications made within the For loop
Return $\mathbf{w}$, b

Fig. 3. The perceptron learning algorithm.

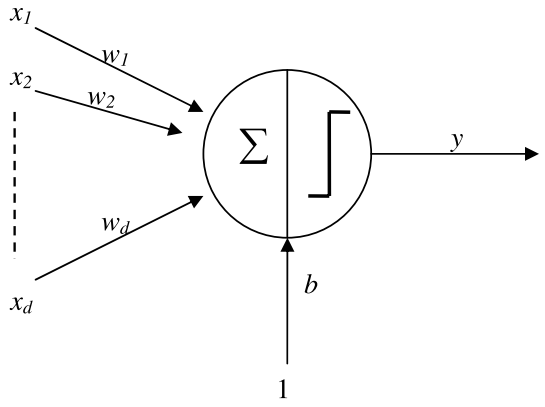

Fig. 4. Perceptron. The output $y = f(\mathbf{x}) = \text{sign}(\mathbf{w}\cdot\mathbf{x} + b) = \text{sign}(\sum w_i x_i + b)$.

layer, at least one middle or hidden layer, and an output layer of computational neurons (see Fig. 5). The input signals are propagated in a forward direction on a layer-by-layer basis.

Learning in a multilayer ANN is similar to learning with a perceptron. That is, each training point $\mathbf{x}_i$ is presented to the network as input. The network computes the corresponding output y. If y and y_i are different, the network weights are adjusted to reduce this error. In a perceptron, there is only one weight for each of the d inputs and only one output. However, in a multilayer ANN, how can we update the weights to the hidden units that do not have a target value? The revolutionary (and somewhat obvious) idea that solved this problem is the chain rule of differentiation (Gershenfeld and Weigend, 1993). This idea of error backpropagation was first proposed in 1969 (Bryson and Ho, 1969) but was ignored due to its computational requirements. The backpropagation algorithm was rediscovered in 1986 (Rumelhart et al. (1986)) at a time when computers were powerful enough to allow for its successful implementation. In the backpropagation algorithm, the error (i.e., the difference between the network output y and the desired output y_i) is calculated and propagated backwards through the network from the output to the input layer. The weights are modified as the error is propagated. The sum of squared errors (over all training data) is the performance measure that the backpropagation algorithm attempts to minimize. When the value of this measure in an entire pass through all training data is sufficiently small, it is considered that the network has converged.

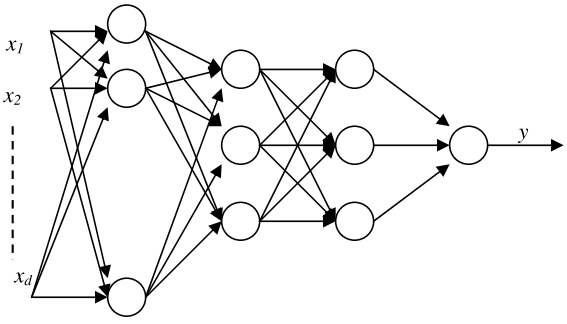

Fig. 5. Multilayer ANN with two hidden layers.

2.2. Support vector machines

A linear decision boundary is a simple classifier that can be learned very efficiently. However, due to its small complexity it can correctly classify data that is linearly separable only. On the other hand, a more complex decision boundary can correctly classify general data that may not be linearly separable. However, such a classifier may be much harder to train. A Support Vector Machine (SVM) combines the best of both worlds. That is, it uses an efficient training algorithm while at the same time being capable of representing complex decision boundaries.

The hyperplanes corresponding to $\mathbf{w}\cdot\mathbf{x} + b = -1$ and $\mathbf{w}\cdot\mathbf{x} + b = 1$ (see Fig. 1) are the bounding hyperplanes. The distance between the two bounding hyperplanes is the margin, which is equal to $\frac{2}{\|\mathbf{w}\|}$. It can be shown that, for given training data, maximizing the margin of separation between the two classes has the effect of reducing the complexity of the classifier and thus optimizing generalization performance. The optimal hyperplane corresponds to the one that minimizes the training error and, at the same time, has the maximal margin of separation between the two classes (Burges (1998)). In order to find the optimal separating hyperplane, the following convex optimization problem is solved

$$\min_{\mathbf{w},b} \frac{1}{2}\|\mathbf{w}\|^2$$

with the constraints that

$$y_i(\mathbf{w}\cdot\mathbf{x}_i + b) \geqslant 1, \quad i = 1, 2, \ldots, n \tag{2}$$

Thus, the task is to maximize the margin while achieving the correct classification of all the training data.

In order to generalize to the case where the decision boundary is not linearly separable, a SVM first maps the data into some other (possibly infinite dimensional) feature space using a mapping $\Phi : \Re^d \to \Re^{d'}$, with $d' \geqslant d$. If the dimensionality of the new feature space is sufficiently high, the data will always be linearly separable. This is because data that is mapped into a sufficiently high-dimensional space will always be linearly separable. Observe that the perceptron learning algorithm (see Fig. 3) works by adding or subtracting misclassified training points to a randomly initialized $\mathbf{w}$. Without any loss of generality, we can assume that $\mathbf{w}$ is initialized to the zero vector and thus its final value will be a linear combination of the training points (Cristianini and Shawe-Taylor, 2000)

$$\mathbf{w} = \sum_{i=1}^{n} \alpha_i y_i \mathbf{x}_i$$

where α_i is a positive value proportional to the number of times misclassification of $\mathbf{x}_i$ has caused $\mathbf{w}$ to be updated.

Intuitively, α_i can also be regarded as a measure of the information content of $\mathbf{x}_i$. The decision function (1) can then be rewritten in dual coordinates as follows (Cristianini and Shawe-Taylor, 2000):

$$\begin{aligned} f(\mathbf{x}) &= \operatorname{sign}(\mathbf{w}\cdot\mathbf{x}+b) \\ &= \operatorname{sign}\left(\left\langle \sum_{i=1}^{n}\alpha_i y_i \mathbf{x}_i \cdot \mathbf{x}\right\rangle + b\right) \\ &= \operatorname{sign}\left(\sum_{i=1}^{n}\alpha_i y_i \langle \mathbf{x}_i \cdot \mathbf{x}\rangle + b\right) \end{aligned}$$

where the α_i's can be found by solving the following dual optimization problem:

$$\max_{\alpha} \sum_{i=i}^{n}\alpha_i - \frac{1}{2}\sum_{i=1}^{n}\sum_{j=1}^{n}\alpha_i\alpha_j y_i y_j \mathbf{x}_i^{\mathrm{T}}\mathbf{x}_{\mathbf{j}} \tag{3}$$

with the constrains that

$$C \geqslant \alpha_i \geqslant 0, \quad i = 1,2,\ldots,n \quad \sum_{i=1}^{n}\alpha_i y_i = 0$$

where C is the soft–hard margin penalty, which represents the tradeoff between the margin size and the number of misclassifications. Those points for which $\alpha_i > 0$ are called support vectors and lie closest to the hyperplanes. All other points have $\alpha_i = 0$ thus the support vectors are the critical elements of the training set. The number of support vectors is usually much smaller than n. The key property of this dual representation is that only the inner products of the training data with the new test point are needed. This implies that there is no need to evaluate $\Phi(\mathbf{x}_i)$ or $\Phi(\mathbf{x}_j)$ as long as we know what the value of $\Phi(\mathbf{x}_i)\cdot\Phi(\mathbf{x}_j)$ is. We can use a kernel function to avoid having to perform an explicit mapping into the higher-dimensional feature space. A kernel function K calculates the dot product in the higher-dimensional feature space of the image of 2 points from the original feature space, $K(\mathbf{x}_i,\mathbf{x}_j) = \Phi(\mathbf{x}_i)\cdot\Phi(\mathbf{x}_j)$. Mercer's theorem (1909) indicates that any kernel whose matrix $\mathbf{K}_{ij} = K(\mathbf{x}_i,\mathbf{x}_j)$ is positive definite corresponds to some feature space and is thus a valid kernel. The importance of this is that we can learn complex decision boundaries in the higher-dimensional feature space efficiently (i.e., without having to work with the higher-dimensional feature space representation of each data point). When mapped back to the original feature space, the resulting linear separators can correspond to arbitrary non-linear decision boundaries between the two classes. Substituting $K(\mathbf{x}_i,\mathbf{x}_j)$ for $\mathbf{x}_i{}^{\mathrm{T}}\mathbf{x}_j$ in (3) gives the following optimization problem:

$$\max_{\alpha} \sum_{i=i}^{n}\alpha_i - \frac{1}{2}\sum_{i=1}^{n}\sum_{j=1}^{n}\alpha_i\alpha_j y_i y_j K(\mathbf{x}_i,\mathbf{x}_j)$$

with the constrains that

$$C \geqslant \alpha_i \geqslant 0, \quad i = 1,2,\ldots,n \quad \sum_{i=1}^{n}\alpha_i y_i = 0$$

Solving for the α's in the above optimization problem results in the following final decision function:

$$f(\mathbf{x}) = \operatorname{sign}\left(\sum_{i=1}^{n} y_i\alpha_i K(\mathbf{x},\mathbf{x}_i) + b\right)$$

which corresponds to a linear hyperplane in the higher-dimensional feature space and an arbitrarily complex decision boundary in the original feature space.

2.3. Sensitivity analysis

Selection of input variables is a critical part in the design of a machine learning algorithm. Each additional input unit adds another dimension and contributes to what is known as the *curse of dimensionality* (Bellman, 1961), a phenomenon in which performance degrades as the number of inputs increases. A dimensionality reduction technique can be used to reduce the number of input variables by keeping only the most important ones (i.e., the ones that allow us to retain as much discriminatory information as possible). In the case of classification, we should aim at keeping input variables that result in large interclass distance and small intraclass variance. It is also desirable to find any correlation between input variables so that any redundant information can be removed. The most common approach to dimensionality reduction is Principal Component Analysis (PCA) (or discrete Karhunen–Loeve transform) (Jolliffe, 1986). A disadvantage of PCA is that, in contrast to the original input variables, the derived dimensions may not have an intuitive interpretation. Another approach is to conduct a sensitivity analysis, which quantifies the importance of each input variable with respect to a particular model. Sensitivity analysis methods estimate the rate of change in the output of a model as a result of varying the input values. The resulting estimates can be used to determine the importance of each input variable (Saltelli et al., 2000). A particularly straightforward method consists in observing the rate of change in the output with respect to direct changes (or perturbations) in the inputs. In Goh and Wong (1991), the Sensitivity Causal Index (SCI) is proposed. Given input vectors $\{\mathbf{x}_i\}_1^n$, where $\mathbf{x}_i \in \Re^d$, for an ANN with single output $y = f(\mathbf{x}_i)$ (see Fig. 5), the SCI for each input dimension or factor x_j is defined as

$$S_j = \sum_{i=1}^{n} |f(\mathbf{x}_i) - f(\mathbf{x}_i + \Delta_{ij})|$$

where $|\cdot|$ denotes absolute value and Δ_{ij} is a small constant added to the jth component x_j of $\mathbf{x}_i$.

3. Methodology

We use machine learning for two purposes: selecting software metrics that are most likely to indicate fault-proneness, and implementing fault-proneness prediction models based on those metrics. The objects whose fault-proneness

is to be predicted are software modules. Given a data set $\{(\mathbf{x}_i, y_i)\}_1^n$, where $\mathbf{x}_i \in \Re^d$ is a vector of software metric values that quantify the attributes of the ith module and $y_i \in \mathbb{N}$ is the number of reported errors, an ANN is trained.

When training an ANN, it is common to normalize each input to the same range. This tends to improve the behavior of the training process, ensures that initial default parameter values are appropriate, and that (initially) every input is equally important. One of the problems with software metrics (e.g., LOC) is that the upper bound of their value range is usually unlimited. This is a problem since, in order to normalize, we need upper and lower bounds for the software metric's value range. It is common to estimate those bounds from the range of values observed in the data set. For each component x_j, we determine the minimum and the maximum values in the data set. Then, the scaled value x'_j, is

$$x'_j = \frac{x_j - \min(x_j)}{\max(x_j) - \min(x_j)}$$

where $\min(x_j)$ and $\max(x_j)$ are the minimum and maximum values, respectively, of the jth component (i.e., software metric) over the n observations in the data set. Thus, each recorded value is mapped to the closed interval $[0, 1]$. Note that this is the relative position of the recorded value within the recorded range of values in the data set. It is important that the data set includes observations from a wide range of potential values for each of the software metrics. We make sure that this is indeed the case in our experimental data. The normalized data set is denoted by $\{(\mathbf{x}'_i, y_i)\}_1^n$.

To train the ANN, a simple holdout cross-validation method is used. That is, the data set is separated into two disjoint sets: the training set and the validation set. The validation set is used for model selection (i.e., to assess generalization performance and determine when to stop the training process so as to avoid overfitting). Notice that if true generalization error estimates were to be computed, an independent test set used only to asses the performance of the final model (i.e., after stopping the training process) would also have to be used. However, the main focus of this paper is on estimating the relative importance of the different software metrics and thus, as long as we are not overfitting the training data, whether or not the estimate of generalization performance provided by the validation set is an underestimate or not is not that critical. The sets are stratified (i.e., the distribution of y_i values in each set is approximately the same as in the entire data set). The training set is used to train the ANN and the validation set is used to test its generalization performance. Then, in order to determine the importance of each software metric in predicting fault-proneness, the SCI is used to perform sensitivity analysis on the trained ANN. The software metrics that are deemed to be the most critical are then used to implement an ANN-based fault-proneness prediction model. Specifically, each $\mathbf{x}'_i$ in the data set is transformed into a (possibly lower-dimensional) vector $\mathbf{x}''_i \in \Re^{d'}$, where $d' \leqslant d$, by removing the software metrics that are considered to be less critical. The reduced data set is denoted by $\{(\mathbf{x}''_i, y_i)\}_1^n$. Then, holdout cross-validation is once again used to train an ANN on the reduced data set and measure its generalization performance.

The fault-proneness prediction problem can also be regarded as a classification task (i.e., a module is predicted to either contain errors or be error-free). We associate a variable c (i.e., the class) with each module in the reduced data set. For each $\mathbf{x}''_i$, the value of c_i is 1 if y_i is at least 1, otherwise it is 0. The classification data set $\{(\mathbf{x}''_i, c_i)\}_1^n$ is used to build ANN and SVM-based classifiers. Similarly, a holdout cross-validation is used to train and test the generalization performance of both the ANN and the SVM.

4. Experiments

4.1. Data description

The data set used in this research is obtained from the NASA IV & V Facility Metrics Data Program (MDP) data repository.[1] The primary objective of the MDP is to collect, validate, organize, store and deliver software metrics data. The repository contains software metrics and associated error data for several projects. The data is made available to general users and has been sanitized by officials representing the projects from which the data originated. For each project in the database, unique numeric identifiers are used to describe product entries. A product refers to anything with which defect data and metrics can be associated. In most cases, it refers to code-related project modules such as functions. For each module, metric values were extracted and mapped to a defect log. Because the recorded metric values for a module correspond to those obtained before eliminating faults in the module, there is no risk of the metric values changing as a result of structural changes in the module that may occur during fault elimination. We use the data associated with the JM1 project. This is a real-time project written in C consisting of approximately 315,000 LOC. There are 10,878 modules. The following 21 software product metrics are associated with each module:

Symbol	Name/description
LOCb	Number of blank lines
LOCc	Number of comment-only lines
LOCe	Number of code-only lines
LOCec	Number of lines which contain both code and comment
LOC	Total number of lines
BR	Number of branches
n_1	Number of unique operators
N_1	Total number of operators
n_2	Number of unique operands
N_2	Total number of operands

Symbol	Name/description
$v(G)$	McCabe's cyclomatic complexity (McCabe, 1976). It is the number of linearly independent paths and measures the complexity of the module's decision structure
$iv(G)$	McCabe's design complexity (McCabe and Butler, 1989). It is derived from $v(G)$ and measures the complexity of a module's calling patterns compared to other modules. It differentiates between modules that will increase a program's design complexity and modules which simply contain complex decision structures
$ev(G)$	McCabe's essential complexity (McCabe, 1976). It measures how much unstructured logic exists in a module
N	Halstead's program length (Halstead, 1977), $N = N_1 + N_2$
V	Halstead's program volume (Halstead, 1977), $V = N * \log_2(n_1 + n_2)$
D	Halstead's program difficulty (Halstead, 1977), $D = \frac{n_1}{2} * \frac{N_2}{n_2}$
L	Halstead's program level (Halstead, 1977), $L = \frac{1}{D}$
E	Halstead's programming effort (Halstead, 1977), $E = D * V$
T	Halstead's programming time (Halstead, 1977), $T = \frac{E}{18}$
B	Halstead's error estimate (Halstead, 1977), $B = \frac{E^{\frac{2}{3}}}{3000}$
I	Halstead's intelligent content (Halstead, 1977), $I = \frac{1}{D} * V$

The number of errors associated with each module is equal to the number of changes due to errors. That is, if a module is changed due to an error report (as opposed to a change request), then it received a one-up count.

4.2. Results

The data set is $\{(\mathbf{x}_i, y_i)\}_1^{10,878}$, where $\mathbf{x}_i \in \Re^{21}$ is a vector of the 21 software metric values that quantify the attributes of the ith module and $y_i \in \mathbb{N}$ is the number of reported errors. For every $\mathbf{x}_i$ in the data set, each of the 21 software metric values is scaled and mapped to the closed interval $[0, 1]$. This scaling ensures that, during training, each of the 21 software metrics is equally important. The resulting normalized data set $\{(\mathbf{x}'_i, y_i)\}_1^{10,878}$, where $\mathbf{x}'_i \in [0, 1]^{21}$ and $y_i \in \mathbb{N}$, is separated into two sets: the training set and the validation set. The sizes of the training and validation sets are 7252 and 3626, respectively. The distribution of y_i values in each set is approximately the same as in the entire data set.

The training set is used to determine values for the free parameters (e.g., number of layers, number of units in each layer, learning rate) and train the ANN. We make no claim on using optimal values for the free parameters as they were selected after a very coarse sampling. The training is stopped at the minimum of the Mean Squared Error (MSE) on the validation set. The MSE is the average error over all samples in the set. Fig. 6 shows the MSE of the ANN on both the training set and the validation set after every 50 training cycles (or epochs). All samples in the training set are presented to the ANN once during each cycle. After 150 training cycles, the MSE on the training and validation sets are 0.7878 and 0.8534, respectively. At that point, the ANN generalizes best. If training is not stopped at that point, overfitting occurs and the generalization performance of the ANN decreases even though the MSE on the training set continues to decrease.

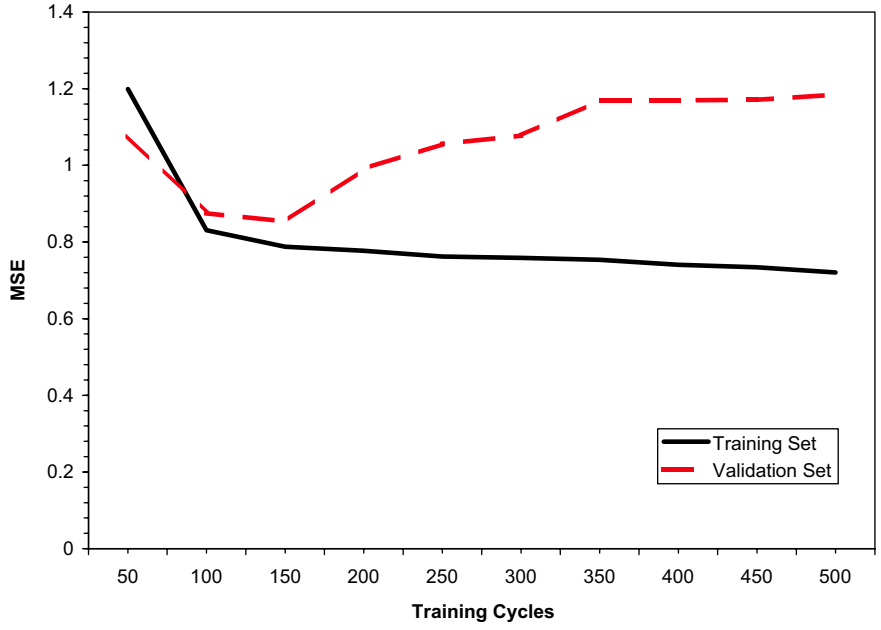

Fig. 6. Normalized data set. Training and validation MSE versus number of training cycles.

The sensitivity analysis of the trained ANN is then performed. The SCI of each input dimension over the entire normalized data set and with $\Delta = 0.1$ is computed. Table 1 shows the average change in the absolute value of the ANN's output in response to perturbations of each of the input variables by Δ (i.e., $\frac{\text{SCI}}{10{,}878}$). The threshold for classifying a software metric as critical or non-critical was also fixed to Δ (i.e., 0.1). That is, a software metric is deemed to be important if, on average, perturbing its value by Δ results in the predicted error value changing by at least Δ. Thus, we keep the software metrics corresponding to the first 14 entries in Table 1 and remove the remaining 7 from the data set. In this way, the reduced data set $\{(\mathbf{x}_i'', y_i)\}_1^{10{,}878}$, where $\mathbf{x}_i'' \in [0,1]^{14}$ and $y_i \in \mathbb{N}$, is obtained. We notice that the resulting model contains software metrics that are representative of all the dimensions of software complexity (e.g., size, internal complexity, readability, documentation) that were present in the original model.

The reduced data set is used to implement an ANN-based fault-proneness prediction model. The training and validation sets contain the same modules as before. The training set is used to determine values for the free parameters and train an ANN. We make no claim on using optimal values for the free parameters as they were selected after a very coarse sampling. The training process is stopped at the minimum of the MSE on the validation set. Fig. 7 shows the MSE of the ANN on both the training set and the validation set after every 50 training cycles. After 300 training cycles, the MSE on the training and validation sets are 0.62743 and 0.6935, respectively. Stopping the training process at that point results in a ANN-based fault-proneness prediction model with an improvement in generalization performance of almost 20% over the original higher-dimensional model. This confirms the efficacy of sensitivity analysis in determining the importance of software metrics.

Table 1
Average sensitivity of each variable in decreasing order (left to right)

Metric	$\frac{\text{SCI}}{10878}$	Metric	$\frac{\text{SCI}}{10878}$	Metric	$\frac{\text{SCI}}{10878}$
LOC	0.5123	LOCe	0.4325	$v(G)$	0.4257
BR	0.3661	n_2	0.3014	N	0.2781
$iv(G)$	0.2397	D	0.2341	N_2	0.1899
V	0.1517	LOCc	0.1477	n_1	0.1407
LOCec	0.1235	E	0.1016	B	0.0906
L	0.0874	N_1	0.0869	LOCb	0.0793
$ev(G)$	0.0513	I	0.0417	T	0.0261

Next, we consider the fault-proneness prediction problem as a classification task. We associate a variable c (i.e., the class) with each module in the reduced data set. For each $\mathbf{x}_i''$, the value of c_i is 1 if y_i is at least 1, otherwise it is 0. The classification data set $\{(\mathbf{x}_i'', c_i)\}_1^{10{,}878}$, where $\mathbf{x}_i'' \in [0,1]^{14}$ and $c_i \in \{0,1\}$, is used to build ANN and SVM-based classifiers. The training and validation sets contain the same modules as before. We make no claim on using optimal values for the free parameters of both the ANN and the SVM as they were selected after a very coarse sampling. A logistic activation function is used for the ANN's output node so that the network output is $y \in [0,1]$ instead of $y \in \Re$. The threshold for classifying modules as faulty or error-free was fixed at 0.5. That is, we consider that the network classifies a module as faulty whenever $y \geqslant 0.5$, otherwise we consider it as error-free. The error measure is the proportion of modules that are classified incorrectly. In the case of the ANN, the training is stopped at the minimum of this measure on the validation set. Figs. 8 and 9 show the MSE and proportion of incorrect classifications respectively on both the training

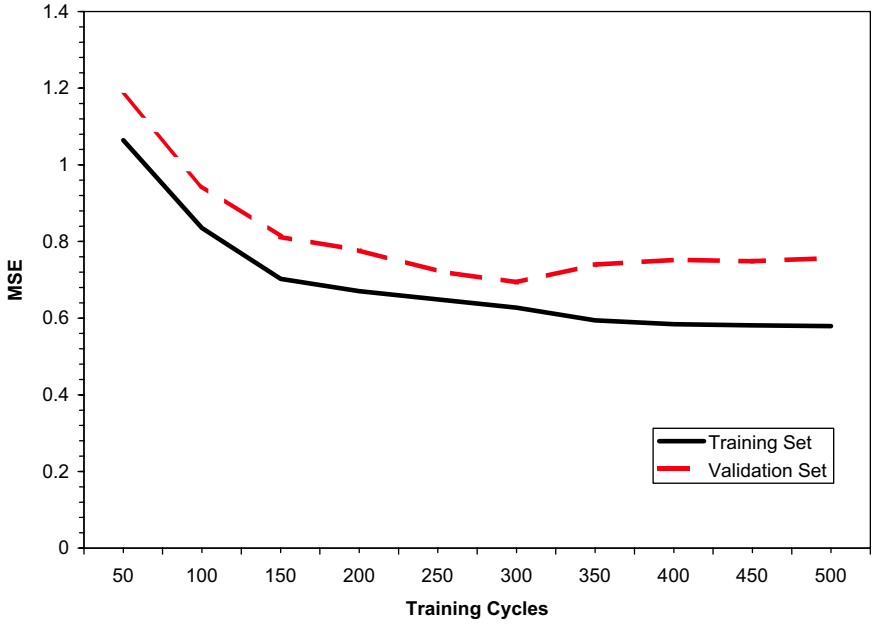

Fig. 7. Reduced data set. Training and validation MSE versus number of training cycles.

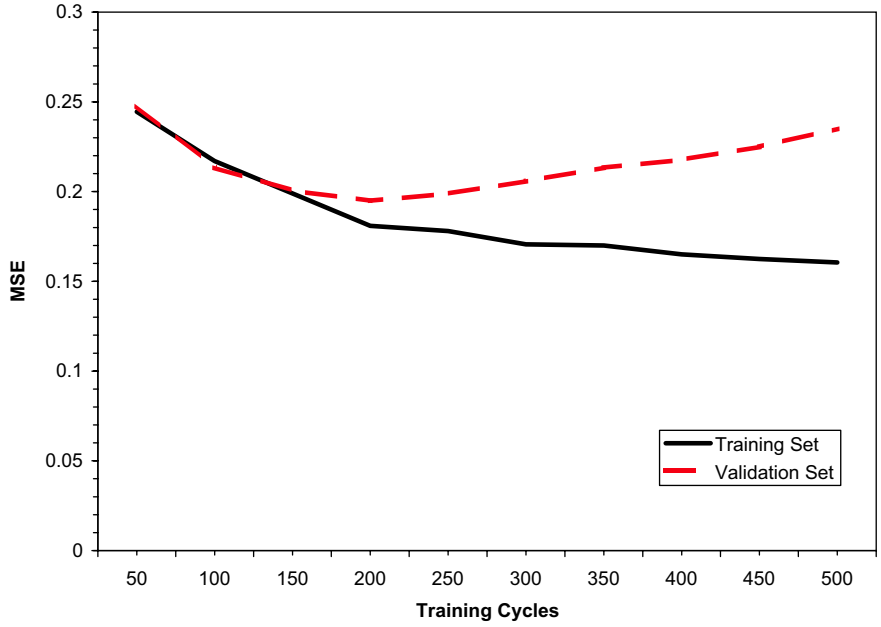

Fig. 8. Classification data set. Training and validation MSE versus number of training cycles.

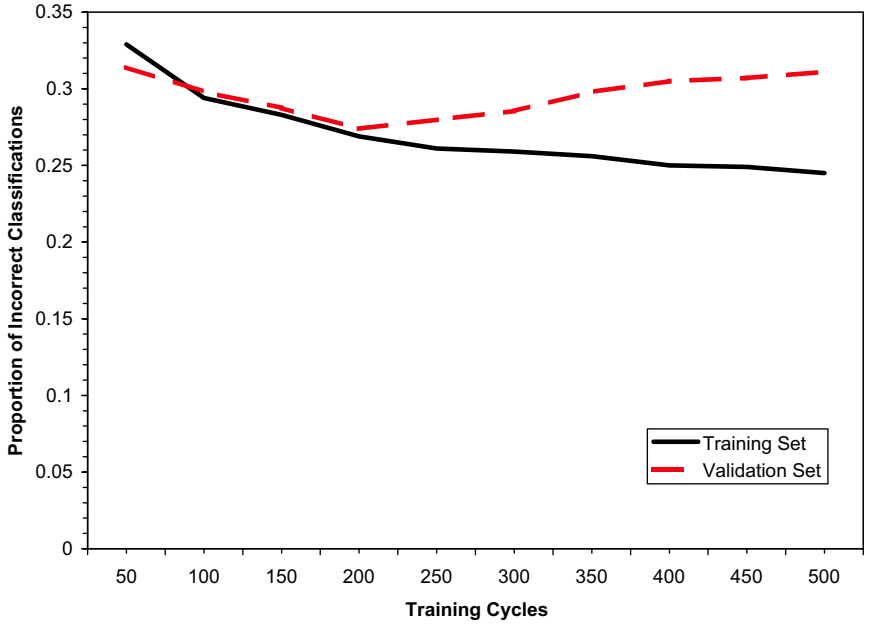

Fig. 9. Classification data set. Proportion of modules classified incorrectly in training and validation sets versus number of training cycles.

and validation sets after every 50 training cycles. After 200 training cycles, the proportion of incorrect classifications on the validation set is 0.2739. That is, 72.61% of the modules are classified correctly.

We use a Gaussian kernel for the SVM. After training the SVM (i.e., solving the SVM optimization problem on the training set), we measure its generalization performance. The percentage of correct classifications on the validation set is 87.4%, which is significantly larger than the 72.61% achieved by the ANN. This confirms the superior performance of SVMs over ANNs when viewing fault-proneness prediction as a binary classification task.

5. Conclusions

This paper investigated the application of machine learning to the estimation of software fault-proneness. Two main goals were fulfilled. First, the use of sensitivity analysis in the selection of software metrics that are more likely to indicate the existence of errors was proposed.

The main advantage of this approach in comparison to other methods such as PCA is that the use of derived non-intuitive metrics is not required. The experimental results provided empirical evidence in support of the effectiveness of this approach. The second goal was to perform a comparison between ANNs and SVMs when applied to the problem of classifying modules as faulty or error-free. The experimental results confirmed the superior performance of SVMs over ANNs when viewing fault-proneness prediction as a binary classification task.

The relationships between software metrics and fault-proneness are often complex. Thus, the adequacy of traditional linear models is compromised. Various publications that use naive methods like regression and correlation between values have been unable to establish a solid relationship between particular software metrics and fault-proneness. Machine learning methods (e.g., ANNs which are capable of inferring complex non-linear input–output transformations) have been proven to be very practical when applied to poorly understood domains. The main drawback of using machine learning in software fault-proneness prediction is the scarcity of data. This is primarily due to the fact that most large corporations are not willing to release their software metric data to the public. The application domain of the software metric repository used in this paper comes from NASA. Thus, as with any empirical data mining study, our conclusions are biased according to the particular data set that was used to generate them. Nevertheless, we argue that the results obtained from the NASA application domain can be generalized to the software engineering industry. This has also been argued by other noted researchers such as Basili et al. (2002). The main reason for this is that NASA makes extensive use of contractors from many other industries including government and commercial organizations. In order to obtain a better understanding of both model and sampling bias, we will have to explore different machine learning algorithms and data sets as part of our future work.

References

Basili, V., McGarry, F., Pajerski, R., Zelkowitz, M., 2002. Lessons learned from 25 years of process improvement: the rise and fall of the NASA software engineering laboratory. In: Proceedings of the IEEE and ACM International Conference on Software Engineering, pp. 69–79.

Bellman, R., 1961. Adaptive Control Processes. Princeton University Press.

Boehm, B.W., 1987. Industrial software metrics top 10 list. IEEE Software 4 (5), 84–85.

Boehm, B.W., Papaccio, P.N., 1988. Understanding and controlling software costs. IEEE Transactions on Software Engineering SE-14 (10), 1462–1477.

Briand, L.T., Basili, V.R., Hetmanski, C., 1993. Developing interpretable models for optimized set reduction for identifying high-risk software components. IEEE Transactions on Software Engineering SE-19 (11), 1028–1034.

Bryson, A.E., Ho, Y.C., 1969. Applied Optimal Control. Blaisdell, New York.

Burges, C., 1998. A tutorial on support vector machines for pattern recognition. Data Mining and Knowledge Discovery 2 (2), 121–167.

Cristianini, N., Shawe-Taylor, J., 2000. An Introduction to Support Vector Machines and Other Kernel-Based Learning Methods. Cambridge University Press.

Gershenfeld, N.A., Weigend, A.S., 1993. Time Series Prediction: Forecasting the Future and Understanding the Past. In: The Future of Time Series. Addison-Wesley, pp. 1–70.

Gill, G., Kemerer, C., 1991. Cyclomatic complexity density and software maintenance productivity. IEEE Transactions on Software Engineering 17 (12), 1284–1288.

Goh, T.H., Wong, F., 1991. Semantic extraction using neural network modeling and sensitivity analysis. In: Proceedings of IEEE International Joint Conference on Neural Networks, pp. 18–21.

Halstead, M.H., 1977. Elements of Software Science. Elsevier, New York.

Jolliffe, I.T., 1986. Principal Component Analysis. Springer.

Khoshgoftaar, T.M., Allen, E.B., 1998. Classification of fault-prone modules: prior probabilities, costs, and model evaluation. Empirical Software Engineering 3 (3), 275–298.

Khoshgoftaar, T.M., Munson, J.C., 1990. Predicting software development errors using complexity metrics. IEEE Journal on Selected Areas in Communications 8 (2), 253–261.

Khoshgoftaar, T.M., Lanning, D.L., Pandya, A.S., 1994. A comparative study of pattern recognition techniques for quality evaluation of telecommunications software. IEEE Journal on Selected Areas in Communications 12 (2), 279–291.

Lehman, M., Perry, D., Ramil, J., 1998. Implications of evolution metrics on software maintenance. In: Proceedings of International Conference on Software Maintenance, pp. 208–217.

Li, H.F., Cheung, W.K., 1987. An empirical study of software metrics. IEEE Transactions on Software Engineering SE-13 (6), 697–708.

McCabe, T.J., 1976. A complexity measure. IEEE Transactions on Software Engineering SE-2 (4), 308–320.

McCabe, T.J., Butler, C.W., 1989. Design complexity measurement and testing. Communications of the ACM 32 (12), 1415–1423.

Mercer, J., 1909. Functions of positive and negative type and their connection with the theory of integral equations. Philosophical Transactions of the Royal Society of London, A (209), 415–446.

Mitchell, T., 1997. Machine Learning. McGraw Hill.

Munson, J.C., Khoshgoftaar, T.M., 1990. Regression modelling of software quality: an empirical investigation. Information and Software Technology 32 (2), 106–114.

Munson, J.C., Khoshgoftaar, T.M., 1992. The detection of fault-prone programs. IEEE Transactions on Software Engineering SE-18 (5), 423–433.

Myrtveit, I., Stensrud, E., Shepperd, M., 2005. Reliability and validity in comparative studies of software prediction models. IEEE Transactions on Software Engineering 31 (5), 380–391.

Neumann, D.E., 2002. An enhanced neural network technique for software risk analysis. IEEE Transactions on Software Engineering 28 (9), 904–912.

Porter, A., Selby, R., 1990. Empirically guided software development using metric-based classification trees. IEEE Software 7 (2), 46–54.

Rosenblatt, F., 1958. The perceptron: a probabilistic model for information storage and organization in the brain. Psychological Review (65), 386–408.

Rumelhart, D.E., Hinton, G.E., Williams, R.J., 1986. Learning representations by back-propagating errors. Nature, 533–536.

Saltelli, A., Chan, K., Scott, E.M., 2000. Sensitivity Analysis. John Wiley & Sons.

Shen, V.Y., Yu, T.J., Thebaut, S.M., Paulsen, L.R., 1985. Identifying error-prone software – an empirical study. IEEE Transactions on Software Engineering SE-11 (4), 317–324.

Xing, F., Guo, P., Lyu, M.R., 2005. A novel method for early software quality prediction based on support vector machine. In: Proceedings of IEEE International Conference on Software Reliability Engineering, pp. 213–222.