Journal of AI and Data Mining
Vol 5, No 2, 2017, 149-167

Evaluation of Classifiers in Software Fault-Proneness Prediction

F. Karimian and S. M. Babamir*

Department of Computer Engineering, University of Kashan, Kashan, Iran.

Received 29 April 2016; Revised 12 June 2016; Accepted 30 October 2016
**Corresponding author: babamir@kashanu.ac.ir (Babamir).*

Abstract
Reliability of a software counts on its fault-prone modules. This means that the less the software consists of fault-prone units, the more we may trust it. Therefore, if we are able to predict the number of fault-prone modules of a software, it will be possible to judge its reliability. In predicting the software fault-prone modules, one of the contributing features is software metric, by which one can classify he software modules into the fault-prone and non-fault-prone ones. To make such a classification, we investigated 17 classifier methods, whose features (attributes) were software metrics (39 metrics), and the mining instances (software modules) were 13 datasets reported by NASA.
However, there are two important issues influencing our prediction *accuracy* when we use data mining methods: (1) selecting the best/most influential features (i.e. software metrics) when there is a wide diversity of them, and (2) instance sampling in order to balance the *imbalanced* instances of mining; we have two imbalanced classes when the classifier *biases* towards the majority class. Based on the feature selection and instance sampling, we considered 4 scenarios in appraisal of 17 classifier methods to predict software fault-prone modules. To select features, we used correlation-based feature selection (*CFS*), and to sample instances, we implemented the synthetic minority oversampling technique (*SMOTE*).The empirical results obtained show that suitable sampling software modules significantly influences the accuracy of predicting software reliability but metric selection does not have a considerable effect on the prediction. Furthermore, among the other data classifiers, *bagging*, *K**, and *random forest* are the best ones when we use the sampled instances for training data.

Keywords: *Software Fault Prediction, Classifier Performance, Feature Selection, Data Sampling, Software Metric, Dependent Variable, Independent Variable.*

1. Introduction

The software fault prediction methods use software *metrics* and *faulty* modules to guess *fault-prone* modules for the next software version. Hereafter, a *software module* indicates an *instance*, and a *software metric* does a *feature*. When we aim to classify software modules into the faulty and non-faulty ones, the software metrics are considered as *predictor* (*independent*) variables (features), and the faulty/non-faulty modules are done as the *outcome* (*dependent*) variable. *Software metrics* measure/quantify software characteristics such as *line of code* (LOC).
The software fault prediction models have been investigated since 1990s. According to [1], the probability of detection (PD) (71%) of the robust fault prediction models may be higher than that for *software reviews* (60%). According to [1], Fagan claimed that inspections can find 95% of defects before testing was not defended at the IEEE Metrics 2002 conference, and this detection ratio was about 60%.One member of the review team may examine 8-20 software lines of code in a minute. Thus, compared to the software reviews, the software fault prediction methods are more cost-effective to recognize software faults. The advantages of a robust software fault prediction are [2]:

- Reach a dependable system;
- Improving test process by concentrating on the fault-prone modules;

- Improving quality by improving test procedure.

Software quality engineering uses various methods and processes for producing high quality softwares. One efficient method is to apply the *data mining* techniques to software metrics for detection of the potential fault-prone modules. Through these techniques, we employed classification to predict program modules as fault-prone (*fp*) or not-fault-prone (*nfp*) [3-5], in which two noteworthy issues, the feature selection and class imbalance problems, were considered.

The class *imbalance* problem is raised if the *fp* instances are much less than the *nfp* ones. The imbalance problem can cause an undesirable conclusion; however, researchers often do not care about it [3, 5]. The efficiency of the software fault prediction models is affected by two significant numbers: (1) software metrics, and (2) software *fp* modules. The software quality prediction model without balancing up classes will not produce efficient fault predictors.

Before investigating the classifier methods to predict the fault proneness of software modules, we presented 4 scenarios consisting of an informed combination of two data preprocessing steps, feature selection (for selecting the important software metrics), and instance sampling (for the class imbalance problem), according to the Shepperd's work [6].

Some researchers have considered the *feature reduction* techniques such as principal component analysis (*PCA*) to improve the performance of the prediction models [7]. We used the correlation-based feature selection (*CFS*) [8] technique to obtain the relevant metrics, and the synthetic minority over-sampling technique (*SMOTE*) [9] for instance sampling (*fp*/*nfp* modules). We used (1) the *SMOTE* technique because it chooses samples through a non-random way [10, 11], and (2) *CFS* because, according to Catal et al. [12], it has a high performance. Considering the use of the *SMOTE* and *CFS* techniques for the selection of samples and features, figure 1 shows our approach.

The process of using feature (metric) selection and instance(module) sampling concurrently gives 4 scenarios, furnishing 4different training datasets for building the prediction models (classifiers).We made the software metric selection on the (1) *sampled* and (2) *original* modules, providing2 different metric subsets. Note that we will obtain *different metrics* if we use a feature selection method on the sampled or original modules. Having selected the features, we dealt with training the prediction model using the original or sampled modules separately. Accordingly, 4 possible scenarios may be considered:

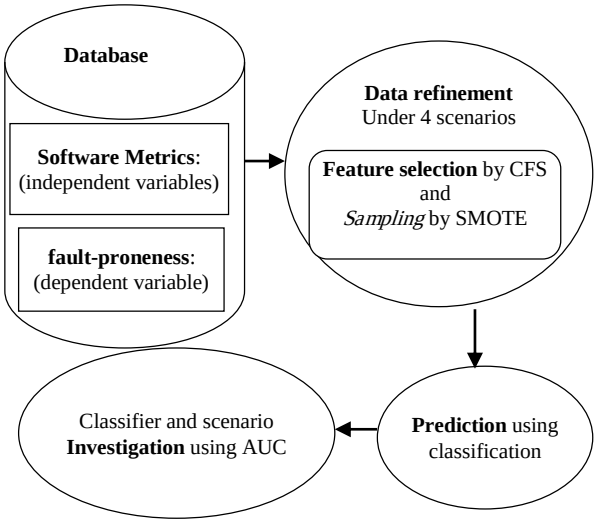

Figure 1. Our approach phases.

- First, to make use of the original modules for feature (metric) selection, and then training the classifiers based on the (**1**) original or (**2**) sampled instances.
- First, to make use of the sampled modules for feature (metric) selection, and then training the classifiers based on the (**3**) original or (**4**) sampled instances.

The research goal of our work was to compare the performance of the fault prediction models based on each of these scenarios, and to detect the best classification model. To this end, we exploited 13 public NASA datasets from PROMISE repository that were created in 2005 [13].

The remainder of this paper is organized as what follows. Section 2 discusses the related works. Section 3 explains the feature selection and sampling methods. Section 4 deals with the classifier methods and techniques applied in this paper. Section 5 deals with our empirical evaluation. Finally, in Section 6, we summarize our conclusions and provide suggestions for the future works.

2. Related works

Various methods have already been applied for software fault prediction. Catal et al. [12, 14, 15] have expanded and validated some artificial immune system-based models in the software fault prediction. Elish et al. [16] have compared the performance of support vector machines (*SVM*s) with the performance of logistic regression, multi-layer perceptron, Bayesian belief network, naive bayes (NB), random forests (RFs), and decision trees, and have finally concluded that the performance of *SVM*s is better

than or (at least the same as) the other methods in the context of 4 NASA datasets. Kanmani et al. [17] have used probabilistic neural network (*PNN*) and back–propagation neural network (*BPN*) with a dataset obtained from the project of graduate students to compare their results with the results of statistical methods. They stated that *PNN* provided a better performance. Gondra [18] has shown that *SVM*s have a higher performance over the artificial neural networks (ANNs)in software fault prediction. Menzies et al. [1] have stated that although it is a very simple algorithm, naive Bayes is the best software prediction model.
The area under the *ROC* (receiver operating characteristic) curve (called *AUC*, and explained in section 5.4) has been applied to evaluate the fault prediction models [3, 5]. Malhotra et al. [19] have shown that, based on *AUC*, LogitBoost is the highest method among the machine learning techniques *ANN*, *RF*, two boosting algorithms (*LogitBoost*, *AdaBoost*), NB, *KStar*, and *bagging* and *logistic regression*; their dataset was some open source software.
Catal et al. [2] have studied machine-learning methods such as RFs and artificial immune systems in the context of public NASA datasets, i.e. the PROMISE repository. They focused on the effects of dataset size, metrics set, and feature selection techniques. They showed (1) RFs had the best prediction performance for large datasets, (2) NB was the best prediction algorithm for small datasets based on *AUC*, and (3) the parallel implementation of artificial immune recognition systems (AIRS2Parallel) was the best artificial immune system paradigm-based algorithm when the method-level metrics were used.
A survey of the feature selection algorithms has been explained in [20]. Typically, the feature selection techniques fall into 2 categories, the *wrapper*-based and *filter*-based approaches. The former trains a learner during the feature selection process, whereas the latter does not depend on training a learner and applies the natural characteristics of instances (which is based on the given metrics) to the feature selection. The latter is computationally faster than the former.
Hall et al. [21] have validated 6 feature selection techniques producing ranked lists of features, and have applied them to 15 datasets in the UCI repository. The experimental results obtained showed that no one approach was the best for all situations. However, if computational complexity is eliminated as a factor, a wrapper-based approach has the best accuracy for a feature selection scheme.
Saeys et al. [22] have perused the use of an *ensemble* of feature selection techniques; this means that the multiple feature selection methods are combined for the feature selection process. They have stated that the ensemble approach provides subset of features more robust than a single feature selection technique.
Khoshgoftar et al. [23] have studied 2 types of feature selections for software defect prediction: (1) individual, and (2) repetitive and sampled with learning processes (boosting *vs.* plain learner).The former denotes that the feature ranking algorithm is used individually on the original data and once. The latter also uses only one feature ranking algorithm but it creates a sample dataset using an under-sampling or over-sampling technique. They applied 6 feature ranking techniques and 2 learners to build classification models (multi-layer perceptron and support vector machine). Their results have shown that the latter enjoys a better performance over the former. Moreover, the ensemble learning (boosting) approach enjoys a better classification performance over the plain learning process, which uses no boosting.
Gao et al. [24] have used 9 filter-based feature-ranking techniques for feature selection with random under-sampling data through 3 scenarios: (1) the features were selected based on the sampled data, and the training data was based on the original data, (2) the features were selected based on the sampled data, and the training data was based on the sampled data, and (3) the features were selected based on the sampled data, and the training data was based on the original data. The SVM classifier was applied to build the classification model, and the eclipse dataset of the PROMISE repository was used. The results obtained demonstrated that the first scenario was better than Scenarios 2 and 3, and the AUC feature-ranking technique performed better than the other approaches.
Similar to their earlier work, Gao et al. [25] applied the three scenarios but they used CFS for feature selection and 5 classifiers (SVM, MLP, LR, KNN, and NB) for constructing the model. The results showed that the 1st scenario performed better than the others, and SVM presented the best performance.
In [26], Gao et al. have applied (1) 6 filter-based feature-ranking techniques before and after the ensemble sampling methods RUSBoost and SMOTEBoost, and (2) 5 different classification algorithms for a group of datasets from real-world software systems. The results obtained demonstrated that feature selection after ensemble sampling was better, and RUSBoost performed

better than SMOTEBoost. Among the 6 ranking techniques, RF (Random Forest) and RFW (random forest walk)enjoyed more performance over the others.

Wang et al. [27] investigated different feature-selection techniques consisting of filter-based and wrapper-based methods, and showed that the efficiency of the classification models improved; however, there was no efficiency when over 85% of the features were removed from the original datasets. The experiments carried out by Catal et al. [12] showed a high performance of the *CFS* method.

The class imbalance problem have been investigated in various areas [28-30], and various techniques have been developed to overcome the difficulties of learning from imbalanced data. In a binary classification, under-sampling the majority class and over-sampling the minority class [31-33] are the main approaches for solving the class imbalance problem. Since in this work, the majority and minority classes are non-faulty and faulty modules, respectively, we used under-sampling the non-faulty module class and over-sampling the faulty module one.

Riquelme et al. [11] have shown that the balancing techniques such as SMOTE improve the *AUC* parameter (see section 3.2.1). They applied the 2 balancing techniques *SMOTE* and resample, with 2 common classification algorithms, NB and J48, on 5 open public datasets from the PROMISE repository. In the current study, we considered the *SMOTE* method to resolve the class imbalance problem in fault prediction modeling.

Considerable works have been done on feature selection and data sampling separately but a few studies have been presented for considering both of them simultaneously, particularly in the software engineering field.

Chen et al. [34] considered data sampling and feature selection in the context of software cost/effort estimation but did not focus on the class imbalance problem, and used data sampling prior to feature selection. Furthermore, their classification model was for non-binary problems.

Liu et al. [35] have introduced the *active* feature selection in their sampling approach. However, their goal of data sampling was dataset size reduction instead of addressing the class imbalance problem.

Khoshgoftaar et al. [36] presented feature selection and data sampling together for software fault prediction. They viewed 6 commonly used feature-ranking techniques [27] for feature selection and the random under-sampling [33] technique for data sampling. However, they used just the *SVM* and *KNN* classifiers for building the software prediction models and their dataset; their results were different from ours. In this paper, we used 4 scenarios consisting of 4 significant synthesis of feature selection and data sampling. Each synthesis was used as a data preprocessing step for the training phase of 17 classifiers in the context of 13 public and cleaned NASA datasets from PROMISE repository.

3. Feature selection and sampling methods

In this section, we address the feature selection and sampling methods exploited in our work.

3.1. Correlation-based feature selection

In machine learning, feature selection is the process of selecting a subset of relevant features for the construction of a prediction model. Instances may contain *redundant* or *irrelevant* features, where the former does not provide additional information and the latter provides no useful information.

Elimination of redundant features from a set of features is called *filtering*. The filtering process may be considered for all or correlated features. For predict models that use machine learning techniques, it is important to determine relevant and significant features. In this work, we used a filter-based method called the correlation-based feature selection (*CFS*) to identify relevant metrics. This method begins with a null set, and at each stage, adds the features having the highest correlation with the class but not have high correlation with the already included features in the set.

3.2. Sampling methods

Sampling is a pre-processing method implemented to balance a given imbalanced dataset by increasing or decreasing the modules (cases) in the dataset before building the prediction model. Usually a dataset consists of a large number of “normal” (unconcerned) instances with just a small number of “abnormal” (concerned) ones. In this work, the normal and abnormal classes were the non-fault prone and fault-prone classes, respectively.

There are 2 types of samplings, *over-sampling* and *under-sampling*. In a binary classification, the former tries to increase the minority (abnormal) class, while the latter tries to decrease the majority (normal) one. Although under-sampling increases the sensitivity of a classifier to the minority class, a combination of over-sampling and under-sampling leads to a better performance

over just under-sampling. Accordingly, prediction of the minority class is improved by correcting the imbalance problem.
Another sampling concern is that over-sampling may lead to over-fitting and under-sampling may lead to elimination of useful instances. Therefore, we used the synthetic minority over-sampling technique (*SMOTE*).

3.3. SMOTE

Chawla et al. [9] have proposed *SMOTE*, producing new instances based on K-nearest neighbor (KNN).To produce sample modules, we used *SMOTE* through the following steps:
1. Normalizing software metrics as the predictor variables. The normalization was used to fit a variable into a specific range. Among the others, the Min-Max normalization maps the metric value $mt_{i,j}$ to $nval(mt_{i,j})$, fitting in the range [0,1] "(1)". The $nval(mt_{i,j})$ value indicates the normalized value for metric mt_i of module j, where $val(mt_{i,j})$ is the current value for the metric of module j, and $min(mt_{i,j})$ and $max(mt_{i,j})$ indicate the max. and min. values for the metric of module j, respectively.

$$\mathrm{nval}\left(\mathrm{mt}_{i,j}\right)=\frac{\mathrm{val}(\mathrm{mt}_{i,j})-\min\left(\mathrm{mt}_{i,j}\right)}{\max\left(\mathrm{mt}_{i,j}\right)-\min\left(\mathrm{mt}_{i,j}\right)} \quad (1)$$

2. Choosing a sample module, say m_s, from the fault-prone class.
3. Computing the KNN value form_s based on the *similarity* (we considered K=5). Among 5 neighbors, the most similar module to m_s is the one that has the least Euclidian distance to m_s. Given that each module consists of n metrics, "(2)" shows the Euclidean distance between m_s and another module (similarity m_s to another module), say m_b, where n indicates the number of metrics of the module.

$$\mathrm{sim}\left(\mathrm{m}_s,\mathrm{m}_b\right)=\sqrt{\sum_{i=1}^{n}[\mathrm{nval}(\mathrm{mt}_{i,s})-\mathrm{nval}(\mathrm{mt}_{i,b})]^2} \quad (2)$$

Having calculated the similarity of module m_s to others, we selected 5 modules having the minimum Euclidian distance to m_s. These modules are called the 5 NNs of module m_s.
4. Choosing one of the 5 neighbors randomly, say m_r, and adding to the minority (fault-prone) class.
5. Generating the *synthetic* module. (a) The difference between each m_s metric value and the corresponding m_r metric was computed as follows (n is the number of metrics):

$$\mathrm{d}_{i,s}=\mathrm{val}(\mathrm{mt}_{i,s})-\mathrm{val}(\mathrm{mt}_{i,r}),\ \mathrm{i=1..n} \quad (3)$$

(b) $d_{i,s}$ is multiplied by a random number between 0 and 1, and added to the corresponding m_s metric value.

$$\mathrm{val}(\mathrm{mt}_{i,s})=\mathrm{val}(\mathrm{mt}_{i,s})+\mathrm{rand}[0,1]*\mathrm{d}_{i,s},\ \mathrm{i=1,n} \quad (4)$$

Step 5 leads to the generality of the decision region of the fault-prone class.

4. Classifiers

We evaluated the statistical and machine learning classifiers [37-39] for software fault prediction. In what follows, we briefly explain them.

4.1. Logistic regression (LR)

LR is widely applied as a statistical technique. A detailed explanation of the LR analysis could be obtained from Hosmer et al. [34] and Basili et al. [11]. It is called *ridge* regression, which is the most commonly used regularization method for the not well-posed problems, meaning that the solution is highly sensitive to changes in data. In this work, we used the multinomial logistic regression model using the *ridge estimator* [40].

4.2. Bagging

Bagging (**b**ootstrap **agg**regat**ing**), introduced by Breiman [41], improves the classification performance using the *bootstrap aggregation*, meaning that it produces various similar sets of training data and applies a new method to each set. It is an *ensemble* classifier, and provides an aggregation of predictions of some independent classifiers with the goal of improving the prediction accuracy. An *ensemble* classifier uses the multiple classification algorithms and averages their predictions. To this end, it uses random samples with replacement and/or random predictor (feature) sets to generate diverse classifications. Therefore, each training set is a bootstrap sample because of using sampling with replacement. The ensemble methods are used to address the *class imbalance* problem.
The bagged classifier makes a decision by the majority of the prediction results returned by each classification. According to [41, 42], the benefits of bagging are (1) a better classification accuracy over the other classifiers,(2) the variance reduction, and (3) avoidance of over-fitting.

4.3. Random forest (RF)

RF was proposed by Breiman [43], and similar to *bagging*, it is an ensemble method. It produces a forest of decision trees at the training time. Each tree is produced based on the values for a random vector; these vectors are sampled with the same distribution and independently for all trees of the

forest. The result of the output class is known as the mode of output classes obtained from the individual trees [42].
According to [41, 42, 44], the features of RF are: (1) simplicity and robustness against noises, (2) ability of accurate classification for various datasets, (3) ability of fast learning, (4) having efficiency on large datasets, (5) ability of estimation of important variables in the classification, (6) ability of estimating missing data and maintaining accuracy in missing a large proportion of the data, and (7) having methods for balancing unbalanced datasets.

4.4. Boosting techniques

Similar to *bagging* and *random forest*, *boosting* is a machine-learning ensemble meta-algorithm. It uses a decision tree algorithm for producing new models. Unlike *bagging*, which assigns an equal vote to each classifier, *boosting* assigns weights to classifiers based on their performance. The *boosting* methods use a training set for each classifier based on the performance of the earlier classifiers.
There are various *boosting* algorithms present in the literature. In this work, we used *AdaBoost*(*AB*) [45] and *LogitBoost*(*LB*) [46] for classification. The features of *boosting* are its ability to (1) reduce bias and variance in supervised learning, and (2) convert *weak learners* to *strong* ones[47]. A weak/strong learner is a classifier that is ill-correlated/well-correlated with the true classification.

4.5. DECORATE

DECORATE (diverse ensemble creation by oppositional relabeling of artificial training examples) is a meta-learner, exploiting a *strong* learner for constructing classes. To this end, *DECORATE* artificially builds random examples for the training phase. This is why *DECORATE* provides a high accuracy on the training data to build efficient various classes in a simple way. The class labels of these artificially constructed examples are in inverse relation to the current classes, and therefore, it increases diversity when a new classifier is trained on the additional data. The problem with the *boosting* and *bagging* classifiers is that they restrict the amount of the ensemble diversity they can get when the training set is small. This is because the *boosting* and *bagging* classifiers provide the diversity by re-weighting the existing training examples, while the *DECORATE* classifier ensures variety on a large set of additional artificial examples.
In the case of class imbalance, identifying samples from the minority class is usually more significant and dearer than the majority class. Therefore, some ensemble methods have been presented to resolve it. According to [48], adding variety to an ensemble method improves the performance of a learning method in case of the class imbalance. Haykin and Network have dealt with the influence of diversity on the performance of the minority and majority classes [48]. They have presented good and bad patterns in imbalanced scenarios, and have obtained 6 different situations of the influence of the diversity through theoretical analysis. Furthermore, they have carried out experimental studies on the datasets consisting of highly skewed class distributions. Then they have come into the conclusion that there is a strong correlation between diversity and performance, and that diversity has a good influence on the minority class.

4.6. Multi-layer perceptron (MLP)

MLP uses biological neurons to construct a model, and is applied to model complex relationships between inputs and outputs and search patterns in datasets [48]. *MLP* could be considered as a binary classifier with multiple layers. An *MLP feed forward* network includes one input layer, one or more hidden layers, and one output layer. Each layer consists of nodes that are connected to their immediate preceding layers as the input and the immediate succeeding layers as the output. The *back-propagation* method is the most commonly used learning algorithm in order to train the multi-layer feed forward networks, and includes 2 passes, forward and backward. Through the forward pass, a training input dataset is used, and a set of outputs is created as the actual response. In this pass, the network weights are fixed and their effect is propagated through the layers of the network [48]. Through the backward pass, an error, which is the difference between the actual and desired output of the network, is computed. The computed error is propagated backward through the network, and the weights are re-adjusted in order to reduce the gap between the actual and desired responses.

4.7. Radial basis function (RBF) network

RBF is a function whose value depends only on the distance (normally the *Euclidean* distance) from the origin. The *RBF* network, proposed by Broomhead and Lowe [49], is an artificial neural network (ANN) applying *RBF* as an activation function. Among others such as function

approximation and time series prediction, the *RBF* networks could apply to the classification.

The RBF networks often consist of 3 layers: input layer, non-linear RBF hidden layer, and linear output layer, where the input layer is a vector of real numbers, $x \in \mathbb{R}^n$, and the output layer is a scalar function of the input vector (Relation 5). In fact, we have $\varphi: \mathbb{R}^n \longrightarrow \mathbb{R}$.

$$\varphi(x)=\sum_{i=1}^{N} a_i \rho(|x-c_i|) \quad (5)$$

where, N is the number of neurons in the hidden layer, c_i is the center vector, and a_i is the weight of neuron i in the output layer. All inputs are connected to each hidden neuron. The RBF network consisting of enough hidden neurons can approximate any continuous function with a desired accurate [50]. The RBF networks could be normalized; in this work, we used the normalized Gaussian RBF network.

4.8. Naïve bayes (NB)

The NB classifier is a probabilistic classifier based on the Bayes theorem, assuming that there is a strong (naive) independency between the features [51]. When *NB* equips with an appropriate preprocessing, it classifies as well as some advanced methods such as the support vector machine (*SVM*).

Instead of the expensive iterative approximation, which is used by many classifiers, the *NB* classifier uses maximum-likelihood (i.e. without Bayesian methods), and training is performed through assessing a closed-form expression in the linear time.

A feature value is independent from the other feature values in the *NB* classifier. Accordingly, the *NB* classifier considers each feature independently in the sample classification, regardless of the correlations with other features of the sample.

The *NB* classifiers can be trained efficiently using the supervised learning for some types of probability models, and their advantage is that they require a small amount of training data for the classification process.

4.9. Bayes network (BN)

A Bayesian network is a probabilistic graphical model that shows relationships among the subsets of variables. Unlike the NB classifier, this method considers dependencies between variables, and determines joint conditional probability distributions. The advantages of a *BN* model are: (1) it easily handles the missing data because of representing dependencies between variables, (2) it could provide a graphical model of causal relationships, and hence could be used to predict the consequences of intervention, and (3) since it has both the causal and probabilistic semantics, it is ideal for incorporating prior knowledge (which typically comes in the causal form) [52].

4.10. Support vector machines (SVM)

SVM, proposed by Vapnik [53], is a supervised learning method creating a hyper-plane or collection of hyper-planes, and can be used for classification and regression. When a hyper-plane has the largest distance to the nearest training data of any class (called functional margin) a good separation is obtained because a larger margin leads to a smaller error of the classifier. *SVM* could be used for the ill-posed problems, meaning that the solution is highly sensitive to the changes in a dataset.

The main problem with *SVM* is that it is not possible to separate the datasets linearly in a finite dimensional space. Accordingly, the original finite-dimensional space is mapped into a higher-dimensional space so that we can separate the datasets [54]. The hyper-planes in the higher-dimensional space are the set of points whose dot product with a vector in that space is constant. Another problem with *SVM* is that despite a good performance in the pattern, recognition field does not consider the problem domain knowledge; moreover, the classification speed is considerably slower than that of the neural networks.

4.11. K*

*K** is an instance-based learning method, using the *entropy distance* to compute the distance between instances [55]. Learning based on instances means that the instance classification is carried out through comparing the instances with a dataset of pre-classified examples. Such a learning is based on the fact that similar instances have similar classifications. The similarity between 2 instances is determined according to a distance function, and a classification function is used to exploit the instance similarity for the classification of the new instances. The entropy distance manages (1) symbolic attributes, (2) real-valued features, and (3) missing values.

4.12. DecisionStump (DS)

DS is a binary classifier, and has a one-level decision tree with one root node connected to the terminal nodes (leaves) [56]. The prediction through DS is carried out based on the value for a single input attribute. *DS* is often used as a

component of the ensemble methods such as bagging and boosting.

4.13. J48

J48 is a Java implementation of the C4.5 algorithm [57], and is a decision tree-based classifier. A decision tree is a machine-learning predictor that predicts the *dependent* variable value based on the attribute values of the existing data. The dependent variable is the attribute that should be predicted. The independent variables are other attributes, which are used to predict the dependent variable value.

A decision tree has internal nodes, indicating different attributes, where the attribute values for the observed samples are shown on branches between the nodes. The final values for the dependent variables are shown by the tree leaves. To classify a new sample, the *J48* decision tree classifier creates a decision tree based on the attribute values of the training dataset. Afterwards, the order of attribute selection is followed based on the tree. The target value of a new instance is predicted through checking values of all attributes against the corresponding values in the decision tree model.

4.14. An alternating decision tree (ADTree)

The *ADTree* classifier combines decision trees with the prediction accuracy of the boosting classifier in a set of classification rules. The *ADTree* classifier consists of *decision* and *prediction* nodes [58], where the former is used to determine conditions and contains both the root and leaf nodes. The latter nodes have a single number. Classifying a sample by an *ADTree* is different from classifying it by the binary classification trees such as *C4.5* because a sample follows only one path in tree in *C4.5*, while in *ADTree*, a sample follows all paths for which the decision nodes are true; then all the prediction nodes visited in these paths were considered. A variation in *ADTree* is the multi-class *ADTree* [59].

4.15. PART

PART is a Java implementation of the C4.5 algorithm [60]. *PART* is a partial decision tree algorithm, applying the divide-and-conquer method, builds a partial C4.5 decision tree in a number of iterations, and adds the best leaf to a rule. The main feature of the *PART* classifier is that it needs no global optimization, while C4.5 does such an optimization.

5. Empirical evaluation

This section aims to represent the empirical study results to evaluate the ability of classifiers in predicting fault-prone software modules. We used the *weka* toolkit with default settings.

5.1. Datasets

We used cleaned versions of the datasets of 13 mission critical NASA software projects (Table 1) in this work; they were available from the PROMISE repository. The software metrics were considered as the *independent variables* (the predictor variables), and the faulty-prone and non-faulty-prone classes were considered as the *dependent variables* (the predicted variables).

Table 1. NASA PROMISE datasets.
Legends: NSM: #software metrics, NI: #instances, %DI: %defective instances (modules).

	Dataset	Language	NSM	NI	%DI
1	CM1	C	21	439	10.47
2	JM1	C	22	7782	20.71
3	KC1	C++	22	1183	21.4
4	KC2	Java	22	334	27.84
5	KC3	Java	40	325	12.92
6	MC1	C & C++	39	1988	1.81
7	MC2	C	40	157	32.48
8	MW1	C	38	379	7.38
9	PC1		22	946	6.65
10	PC2		37	1391	1.50
11	PC3	C	38	1436	10.44
12	PC4		38	1287	13.67
13	PC5	C++	39	1711	26.82

5.2. Independent variables

We considered 39 software metrics as the independent variables. They were quantitative values indicating the software features. The metrics are explained briefly below, and are of 3 types: (1) *module-level* called *McCabe* metrics [61, 62], (2) *Halstead*, and (3) *enumerated* metrics.

The *module-level* metrics consisting of metrics 1-4, 24, 26, 28, 30, 31, 33, and 37 were considered using flow-graph of a module, the *Halstead* metrics consisting of metrics 6-12 and 32 were used for the experimental verifications of a module, and the *enumerated* metrics consisting of metrics 5, 13-23, 25, 27, 29, 34-36, 38, and 39 indicate the number of comments, instructions, delimiters, and blank lines of a module. The

abbreviations used at the beginning of the metrics are used by the PROMISE dataset.

Loc: total number of lines
1. v(g): cyclomatic complexity= *p*+1, where *p* denotes the predicate (branch) of the module;
2. ev(g): essential complexity, denoting unstructured codes of a module, and used to compute the *effort* prediction for the module maintenance;
3. iv(g): design complexity: number of calls directly performed by a module or number of modules directly call a module*;*
4. n: parameter count: number of parameters of a module;
5. v:volume = length× $\log_2(\eta_1+\eta_2)$,where η_1 and $\eta_2$denote the number of distinct operators and operands of a module, respectively;
6. l:length = $N_1 + N_2$, where N_1 and N_2denote the total number of operands and operators of a module, respectively;
7. d: *difficulty* = $(\eta_1/2)^* (N_2/\eta_2)$, parameters η_1, N_2, and η_2 were explained above. This metric denotes the module understanding;
8. i: *content = level*×*volume*, where program *level* ranges between zero and one, and *level*=1 denotes that a module has been composed at the highest possible level (i.e. with a minimum size);
9. e: effort = difficulty×*volume;* the effort estimated for development of a module; *difficulty* is computed as *D*=1/*level*. As the module volume increases, its level and difficulty decreases and increases, respectively;
10. error_est: error estimation=*(*effort$^{2/3}$*)/3000;* the number of errors is estimated to code a module;
11. prog_time: effort/18 seconds; the required time to program;
12. LOCode: number of instructions of a module;
13. LOComment: number of comment lines of a module;
14. LOBlan: number of blank lines of a module;
15. uniq_op:number of unique operators of a module;
16. uniq_opnd: number of unique operands of a module;
17. total_op: total number of operators of a module;
18. total_opnd: total number of operands of a module;
19. branch_count: number of branches of a module;
20. call_pairs: number of invocations by a module;
21. loc_code_and_comment: number of instructions and comment lines of a module;
22. condition_count: number of condition points of a module;
23. cyclomatic_density = v(g) / (LOCode + LOComment);
24. decision_count: number of decision points of a module;
25. design_density:iv(g)/*v*(g);
26. e: edge_count: number of edge flow graph of a module;
27. essential_density: (*ev*(g)-1)/(*v*(g)-1);
28. loc_executable: number of lines of executable code of a module;
29. gdv(g):global_data_complexity = v(g)/*n* (see Parameter 4 for *n*);
30 global_data_density: *gdv*(g)/*v*(g);
31. L: *halstead_level* = $2^*\eta_2/(\eta_1^*N_2)$;
32. maintenance_severity =ev(g)/*v*(g);
33. modified_condition_count: effect of changing a condition on a decision outcome;
34. multiple_condition_count: number of multiple conditions of a module;
35. node_count: number of nodes of flow graph of a module;
36. normalized_cylomatic_complexity: *v*(g)/*loc*;
37. number_of_lines: number of lines of a module;
38. percent_comments: percentage of comment lines of a module.

The PROMISE calculated metrics 1-20 for datasets 1-4 and 9 in table 1, and all metrics for datasets 5-8 and 10-13. However, because some independent variables might be highly correlated, we used a correlation-based feature selection technique (CFS) [8] to select the best predictors of the original and sampled data (Table 2).

Table 2. Metrics selection using CFS method for original and sampled data.

#	Dataset	Selected metrics of original data	Selected metrics of sampled data
1	CM1	1-4-9-14-16-17	3-4-9-13-14-15-16
2	JM1	1-4-9-13-14-15-22-16-17	3-4-14-15-22-16
3	KC1	6-7-8-9-14-15-17-20	3-4-14-15-17
4	KC2	1-3-9-15-22-18-19	2-3-9-13-14-15-22-17-19
5	KC3	7-22-15	22-31-30-6-37-39
6	MC1	21-38-39	15-21-22-14-26-3-30-31-33-38-39
7	MC2	15-14-4-28-31-8-10-11-36-18	14-26-4-3-31-8-10-11-7
8	MW1	15-14-4-27-9-32-34-36-17	15-21-14-24-4-26-27-35-18-38
9	PC1	9-14-22-15-17	3-4-9-14-22-15-19
10	PC2	20-22-14-26-9-8-18-39	14-4-26-29-33-16-39
11	PC3	15-22-14-9-33-37-17	1-21-22-14-25-3-5-9
12	PC4	22-23-3-39	3-24-26-4-28-5-39
13	PC5	21-22-24-4-3-30-9-8-10-34-16-36-1	15-21-3-24-4-3-5-30-31

5.3. Dependent variable

This work focuses on the prediction of being fault-prone a module. Therefore, our dependent variable was a *boolean* variable consisting of true or false values, indicating that the module was fault-prone (*fp*) or non-fault-prone (*nfp*). Predicting the number of faults is a possible future work if such data is accessible.

5.4. Performance metrics

Since the area under the *ROC* (receiver operating characteristic) curve (called *AUC*) is used to evaluate the fault prediction models (classifier methods) [3, 5], we used *AUC* in this work. The *ROC* curve shows *sensitivity* against *specificity*, where the *sensitivity* and *specificity* denote the probability of true fault detection and the probability of false alarm, respectively. We did not have a good performance when the *AUC* value was less than 0.7. (1-.9 = excellent, .9-.8 = good, .8-.7 = fair, .7-.6 = poor, less than .6 = fail).

If the *fp* and *nfp* modules are regarded as the positive and negative cases, the *ROC* curve will show rates of the *true positive* (i.e. correct prediction in fault-proneness of a module) against the false positive (i.e. incorrect prediction of a non-fault-prone module as a fault-prone module). An *ROC* curve shows the classifier performance, lying between 0 and 1 (the value 1 indicates a *perfect* classifier) [47].

5.5. Environment setting

The parameters of the experimental environment were set for the classifiers, as follow:

Logistic Regression: (1) maxIts=-1 (maximum number of iterations to be performed. Value -1 means until convergence), (2) ridge=10^{-8} (ridge value in the log-likelihood).

Bagging: (1) classifier: RepTree, (2) bagSizePercent=100 (Size of each bag, as a percentage of the training set size).

Random Forest: (1) maxDepth=0 (maximum depth of the trees, 0 for unlimited), (2) numFeatures=0 (number of attributes to be used in random selection, zero means $\log_2$ (number_of_attributes) + 1 is used), (3) numTrees=10 (number of trees to be generated).

Boosting: (1) classifier: DecisionStump, (2) likelihood Threshold=-1.7976931348623157E308 (threshold on likelihood improvement), (3) numRuns = 1 (number of runs for internal cross-validation), (4) weightThreshold=10 (weight threshold for weight pruning).

Decorate: (1) artificialSize=1.0 (number of artificial examples to use during training), (2) classifier: J48, (3) desiredSize=15 (number of classifiers in this ensemble. Decorate may terminate before the size is reached (depending on the value for num Iterations), (4) numIterations=50 (maximum number of iterations to be run).

Multilayer Perceptron: (1) hiddenLayers = (attribs+ classes)/2, (2) learningRate=0.3 (amount that the weights are updated), (3) momentum=0.2 (momentum applied to the weights during updating).

Radial Basis Function: (1) clusteringSeed=1 (random seed to pass to K-means), (2) minStdDev=0.1 (minimum standard deviation for clusters), (3) numClusters=2 (number of clusters for K-Means to be generated), (4) ridge=10^{-8} (ridge value for logistic or linear regression).

Naïve Bayes: (1) KernelEstimator=false (kernel estimator for numeric attributes rather than a normal distribution), (2) SupervisedDiscretization =false (supervised discretization to convert numeric attributes to the nominal ones).

Bayes Network: (1) Estimator= SimpleEstimator (Estimator algorithm for finding the conditional probability tables of the Bayes network), (2) search Algorithm=k_2 (selected method to search the network structures).

Support Vector Machine: (1) c=10 (complexity parameter C), (2) epsilon=0.001 (epsilon for round-off error), (3) kernel: radial basis function(kernel to be used).

*K**: (1) entropicAutoBlend=false (entropy-based blending is not used), (2) globalBlend=20 (parameter for global blending), (3) missingMode: average column entropy curves (to determine how missing attribute values are treated).

J48: (1) confidenceFactor=0.25 (confidence factor for pruning), (2) minNumObj=2 (minimum number of instances in per leaf), (3) subtreeRaising=true (subtree raising operation is considered in pruning).

ADTree: (1) numOfBoostingIterations=10 (number of boosting iterations to be performed), (2) saveInstanceData=false (tree does not save instance data), (3) searchPath: expand all paths (type of search to be performed when it builds the tree. It will do an exhaustive search).

PART: (1) minNumObj=2 (minimum number of instances per rule), (2) confidenceFactor=0.25.

5.6. Cross-validation

A 10-fold cross-validation [63] was used to validate the prediction models. Each dataset was randomly partitioned into 10 folds of the same size.

For 10 times, 9 folds were selected to train the models, and the remaining fold was used to test

the models, with each time leaving out a different fold. All the preprocessing steps (feature selection and data sampling) were done on the training dataset. The processed training data was then applied to build the classification model, and the resulting model was used for the test fold. This cross-validation was repeated 10 times; each fold was used exactly once at the test data.

5.7. Discussion of results

In this section, we aim to show the effect of feature selection techniques in combination with data sampling using 4 scenarios. The scenarios (see Figure 2) include all the possible situations when feature selection and data sampling are used simultaneously to create the training dataset.

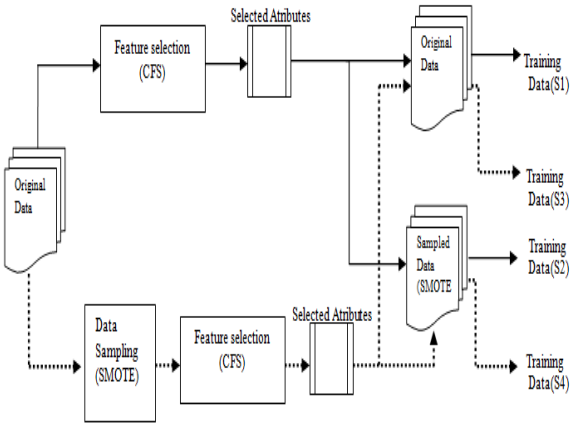

Figure2. Feature selection and data sampling scenarios.

Scenario 1 (S1): using CFS, we select features from the ***original*** data, and create the training dataset based on the ***original*** data;
Scenario 2 (S2): using CFS, we select features from the ***original*** data, and create the training dataset based on the ***sampled*** data;
Scenario 3 (S3): using CFS, we select features from the ***sampled*** data, and create the training dataset based on the ***original*** data;
Scenario 4 (S4): using CFS, we select features from the ***sampled*** data, and create the training dataset based on the ***sampled*** data.

5.7.1. Investigation using AUC performance

As stated in Section 5.4, the *AUC* value denotes the method performance. In other words, if *h* and *g* are 2 classifiers, then AUC(*h*) > AUC(*g*) means that classifier *h* has a better average performance over classifier *g*.
We classified the 13 datasets stated in table 1 using 17 classifiers and the 4 scenarios, and then calculated the *AUC* values. For brevity, we showed just the results of classifying the datasets MC1, MC2, JM1, and KC2 (the datasets 6, 7, 2, and 4 in Table 1) obtained by 17 data classifiers under the 4 scenarios.

According to table 1, MC1 consists of the Min. percent of faulty modules (1.81%), many metrics (39 from 40), and comparatively many instances (1988).
Against MC1, MC2 consists of the max. percent of faulty modules (32.48%) and the min. number of instances (157). Similar to MC1, MC2 contains the max. number of metrics (40). Against MC2, JM1 has the max. number of instances (7782) and the nearly min. numbers of metrics (22 from 40).
Finally, KC2 lacks about half of the metrics (18) and comparatively, does not have many instances (324).
We then proceeded to evaluate the 17 classifiers for classifying the 4 datasets mentioned above. Figures 3-6 show the evaluation under the 4 scenarios for the datasets MC1, MC2, JM1, and KC2. The figures show two issues: (1) method performance with the 4 scenarios, and (2) comparison between the performances of the methods.
The 1st issue shows that with scenarios 2 and 4, we have a higher AUC (performance) than scenarios 1 and 3. The 2nd issue shows that there are agreements and disagreements on the performance of the methods (Table 3). This table shows that the figures agree on the best performance of the *random forest* and *LogitBoost* classifiers (predictors), and the worst performance of *SVM* and *LADTree*.
With the 4 scenarios, we obtained 4 different AUC performance values for each classifier in the classification of the 13 datasets. Then we calculated the *mean* (average) values for the 4 ACU values obtained and *standard deviation* of the scenarios from the mean (Relation 6, μ is the mean) for each classifier.

$$s=\sqrt{\frac{1}{4}\sum_{1}^{4}(\text{scenario}_i\text{-m})^2} \qquad (6)$$

A low *mean* (less than 0.5) and a high *standard deviation* indicate inappropriate values.
A *low* standard deviation means that the data is very close to the mean, while a *high* standard deviation does that data scatter over a wider range of values.

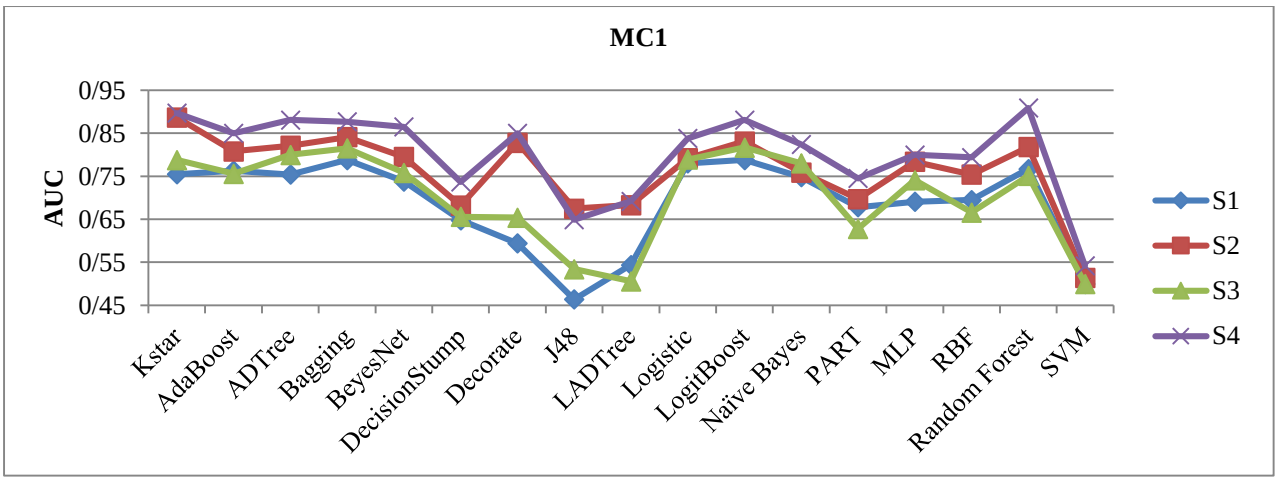

Figure 3. Performance evaluation of 17 data classifiers for MC1 dataset.

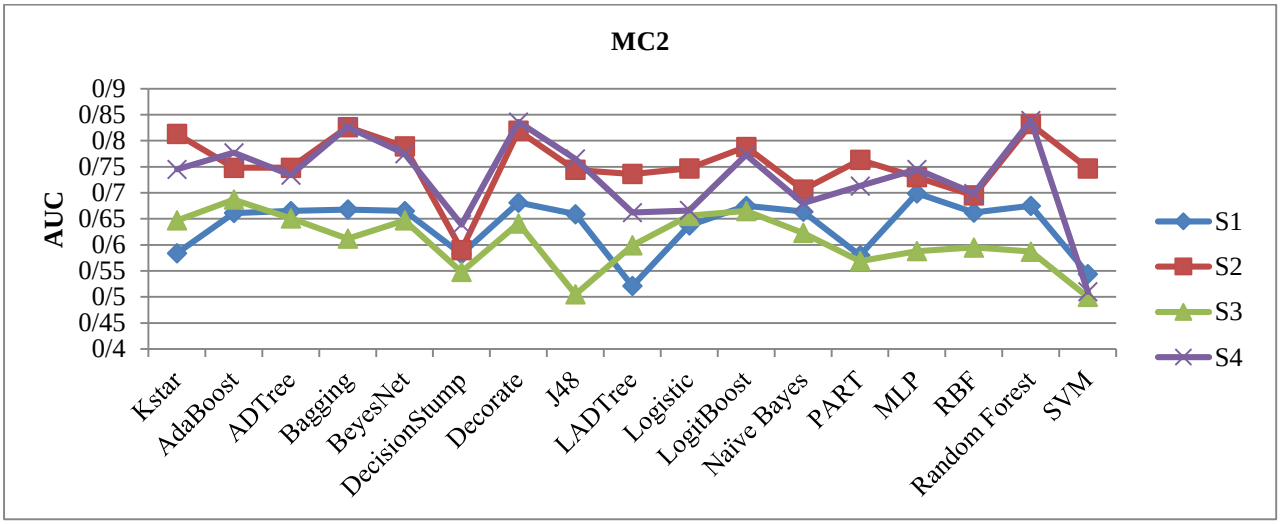

Figure 4. Performance evaluation of 17 data classifiers for MC2 dataset.

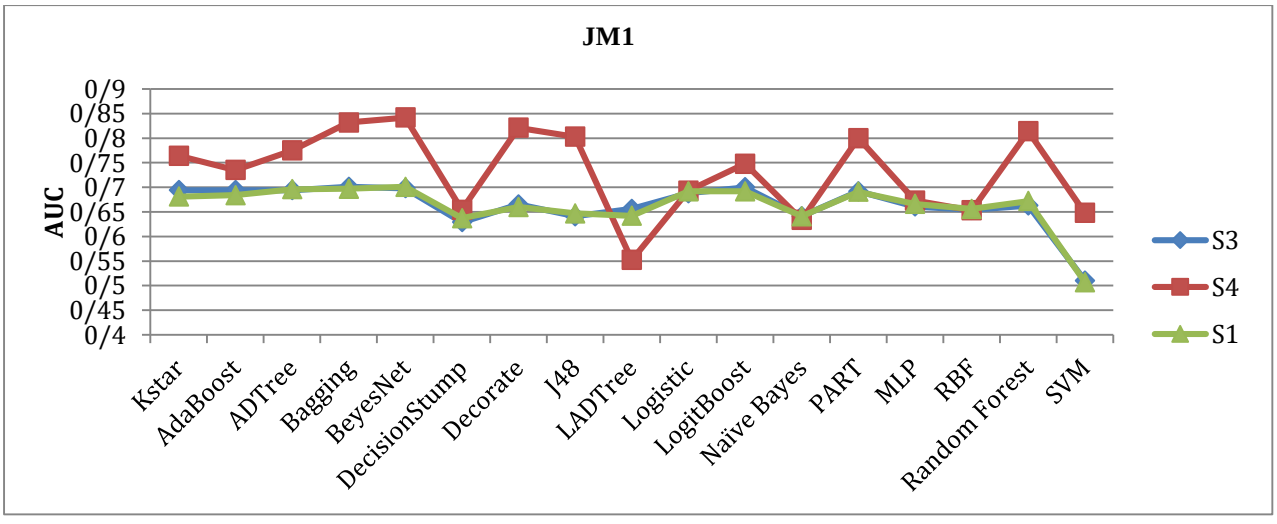

Figure 5. Performance evaluation of 17 data classifiers for JM1 dataset.

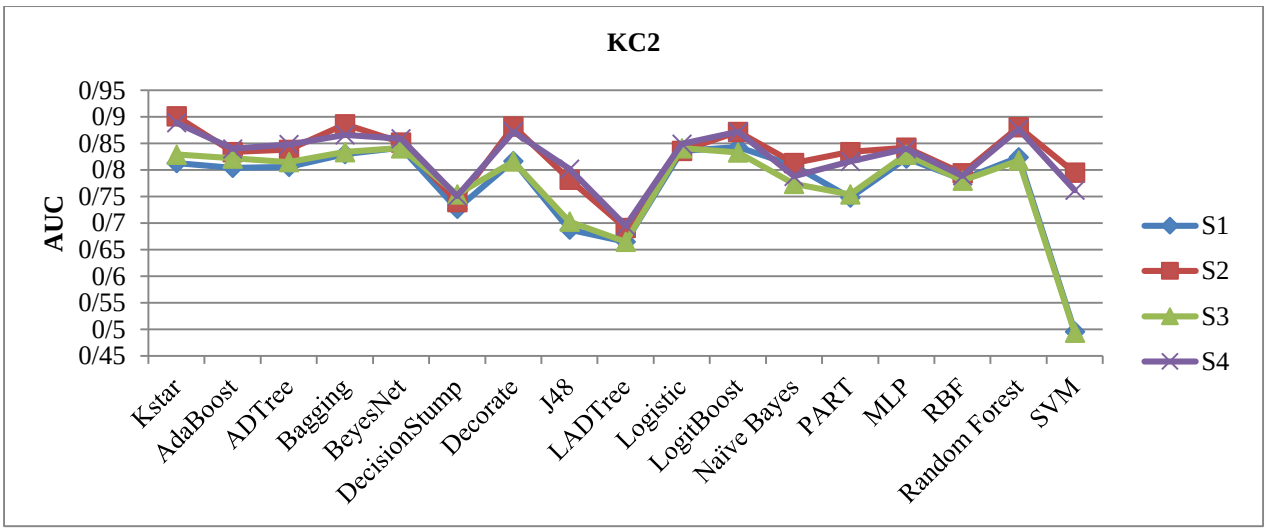

Figure 6. Performance evaluation of 17 data classifiers for KC2 dataset.

Table 3. Performance of classifiers under scenarios.

Dataset	Best classifier under scenario 2/4	Worst classifier under scenario 2/4
MC1	Random Forest, LogitBoost	SVM
MC2	Random Forest, logitBoost	SVM DecisionStump
JM1	BayesNet, Decorate, Random Forest	LADTree
KC2	KStar, Bagging, Decorate, LogitBoost, Random Forest	LADTree

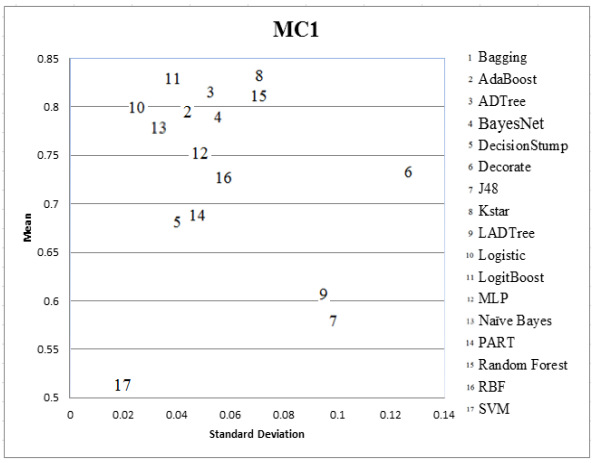

Figure 7. AUC mean value with 4 scenarios and deviation from mean value in *MC1* dataset classification.

Figure 7 shows (1) the mean AUC value obtained with 4 scenarios, and (2) standard deviation of scenarios from the mean for the 17 classifiers in the classification of the *MC1* dataset. According to figure 7, the *worst* (least) mean of the AUC value with the 4 scenarios is that of the *SVM* predictor; therefore, totally, it is not a good prediction method for the MC1 dataset, while the *best* (most) mean of the AUC value with the 4 scenarios is that of *LogitBoost*, *Bagging* and *Kstar*. However, the deviation in the *SVM*, *Logistic*, *Naïve Bayes*, *LogitBoost*, and *Bagging* classifiers from the mean is the lowest; this means that they are stable against the *class imbalance* problem. By contrast, the deviation of the *Decorate*, *J48*, and *LADTree* classifiers from the mean is the *worst* (most). This means that they are more *unstable* for the *class imbalance* problem over others. Overall, figure 7 shows that *Logistic*, *Bagging*, and *LogitBoost* are better predictors than the others with view of the mean and deviation.

Figure 8 shows the mean AUC value obtained using the 4 scenarios, and the standard deviation of the scenarios from the mean for the 17 classifiers in the classification of the *MC2* dataset.

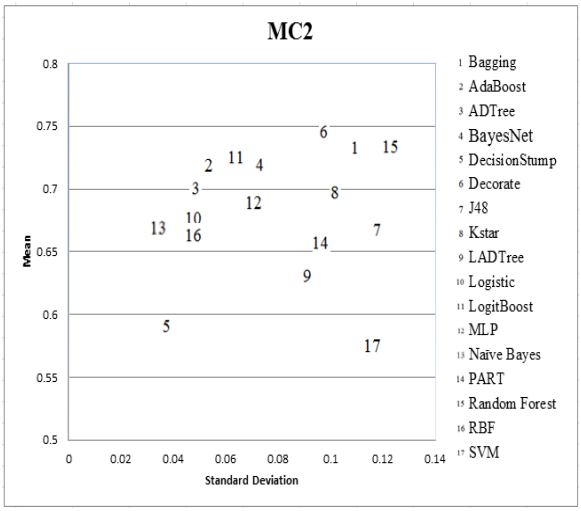

Figure 8. AUC mean value with 4 scenarios and deviation from mean in *MC2* dataset classification.

Similar to the classification used for the *MC1* dataset, Figure 8 shows that the *worst* (least) value of the mean AUC with the 4 scenarios is that of the *SVM* predictor; therefore, totally, it is not a good prediction method for the MC2 dataset; by contrast, the *best* (most) value of the mean AUC

with the 4 scenarios is that of *Decorate*, *Bagging*, and *Random Forest*. However, the deviations in *Naïve Bayes* and *Decision Stump* from the mean are the least. This means that they are stable against *imbalanced* data. Overall, figure 8 shows that *ADTree* and *AdaBoost* predict better than the others with view of the mean and deviation.

Figure 9 shows the AUC mean value obtained with the 4 scenarios and standard deviation of scenarios from the mean for the 17 classifiers used for classification of the *JM1* dataset. Similar to the classification used for the *MC1* and *MC2* datasets, figure 9 shows that the *worst* (least) mean AUC value with the 4 scenarios is that of the *SVM* predictor. Therefore, totally, it is not a good prediction method for *JM1*; moreover, deviation in the *SVM* classifier from the mean value is the worst (most). By contrast, the *best* (most) mean AUC value with the 4 scenarios is that of the *BayesNet*, *Bagging*, and *Part* predictors.

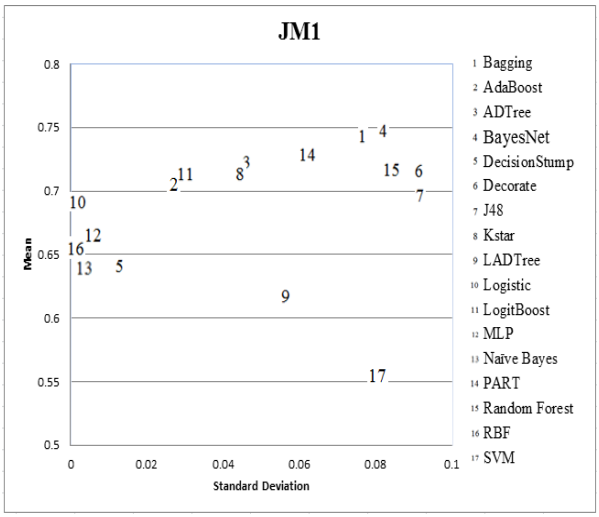

Figure 9. AUC mean value with 4 scenarios and deviation from mean in *JM1* dataset classification.

Deviation of the *Logistic*, *RBF*, *MLP*, *Naïve Bayes*, and *DecisionStump* predictors from the mean value is the least (best).This means that they are stable against the class *imbalance* problem.

Figure 10 shows the mean AUC value obtained with the 4 scenarios and standard deviation of scenarios from the mean for the 17 classifiers used for classification of the *KC2* dataset.

Again, similar to the three previous experiences, Figure 10 shows that the *worst* (least) mean AUC value with the 4 scenarios is that of the *SVM* predictor; therefore, totally, it is not a good prediction method for *KC2*; moreover, similar to the *JM1* dataset, deviation of *SVM* from the mean value is the worst.

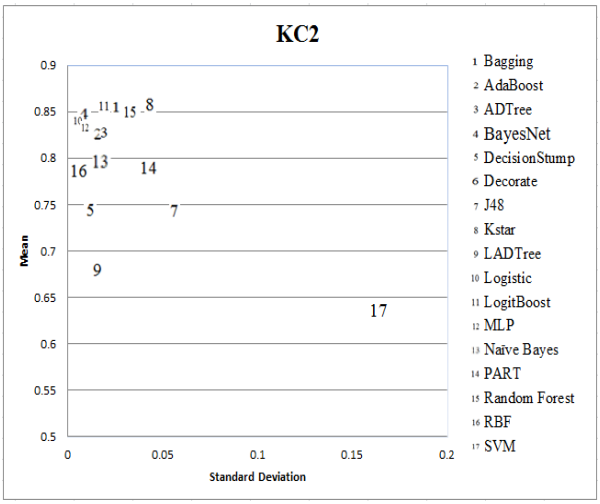

Figure 10. AUC value mean with 4 scenarios and deviation from mean value in *KC2* dataset classification.

However, the mean AUC value of about 10 classifiers is high, and their deviation from the mean value is low. Moreover, the deviation of all classifiers but *SVM* from the mean is *low*. This means that all predictors but *SVM* are stable against the *imbalanced* data.

5.8. Overall evaluations

Figures 11 and 12 show the AUC values obtained for the classifiers in classifying all the datasets with: (1) scenarios 2 and 4, and (2) all the scenarios, respectively.

Furthermore, the figures show the deviation of the classifiers from the mean value. With scenarios 2 and 4, figure 11 shows that the best performance is that of *Bagging* and *Random Forest*, while the worst performance is that of the *SVM* classifier.

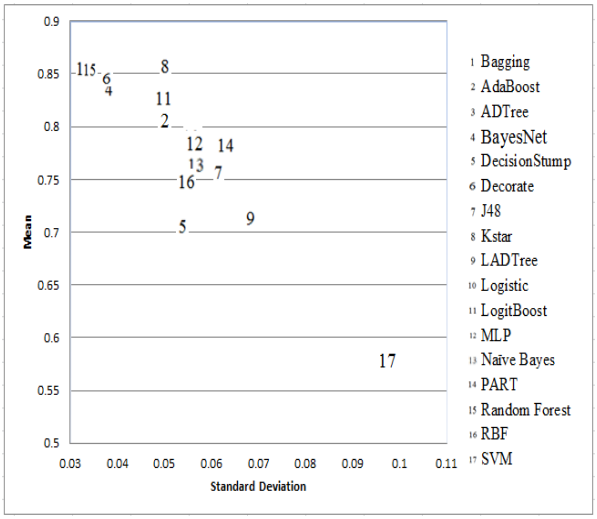

Figure 11. AUC mean value with scenarios 2 and 4 and deviation from mean for classification of *all* datasets.

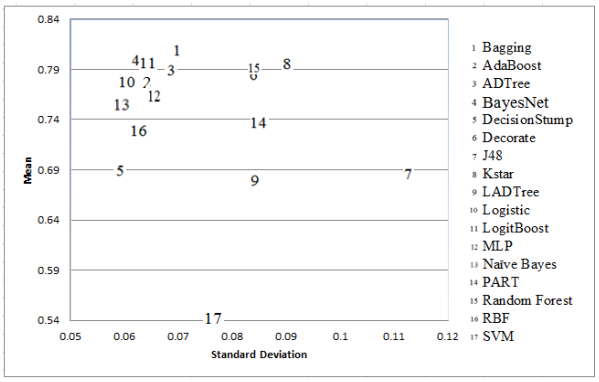

Figure 12. AUC mean of all scenarios and deviation from mean for classification of *all* datasets.

Based on all scenarios, figure 12 shows that the best performance is that of the *BayesNet*, *LogitBoost*, and *Bagging*, while the *worst* one is that of the *SVM* predictor. Moreover, the deviation of the *J48* classifier from the mean value is the *worst*. Based on Figs. 10 and 11, we came into this conclusion that the performance of the classifiers for software fault prediction is according to table 4.

Table 4. Software fault prediction performance of classifiers.

Scenario	Value	Method Performance
2 and 4	Mean	Kstar> Bagging > RF > Decorate >BayesNet
1,2,3,4	Mean	Bagging >BayesNet>LogitBoost>Kstar> RF
1,2,3,4	Standard Deviation	DecisionStump> Naïve Bayes >BayesNet

To show the impact of the 4 scenarios on the prediction models, we classified all of the 13 datasets to faulty and non-faulty modules using the 17 classifiers, and obtained the mean value of the classifiers for each scenario. Then we calculated the impact mean value of the scenarios and the deviation of each scenario from the mean value.

We used one-way analysis of variance (ANOVA) F-test [64] to determine the statistical difference between the 4 scenarios. There are 2 possible hypotheses: (1) null hypothesis, meaning that means of all groups of the population (scenarios) are the same, and (2) alternate hypothesis, meaning that at least one pair of mean values are different.

To show the statistical difference between the mean values of groups of population, the significance level (indicated by probability value or p-value) was computed by ANOVA. The difference between some of the means are statistically significant if p-value≤0.05. Otherwise, we have not enough evidence to reject the null hypothesis, meaning that the means are equal. Therefore, for p-value≤0.05, we concluded that the alternate hypothesis should be accepted, and the means of at least 2 scenarios are significantly different from each other. We used MATLAB [65] to compute the p-value (Table 5).

Table 5. ANOVA results for 4 scenarios.

Source	sum of squares	degree of square	mean sum of squares	F-test	p-value
Scenario	0.08573	m-1= 4-1=3	0.08573/3 =0.02858	0.02858/ /0.00489 =5.84	P[F(3,64) ≥ 5.84] <0.0014
Error	0.31313	n-m= 68-4=64	0.31313/64 =0.00489		
Total	0.39886	64+3=67	0.00489 + 0.02858		

The p-value=0.0014 indicates that the scenarios are different. *Source* means "the source of the variation in the data" and *Scenario* shows groups of the population whose p-values to be compared. *Error* means "the variability within the groups" or "unexplained random error." Parameters *m=4* and *n=68* denote the number of scenarios and data, respectively.

After rejecting the null hypothesis, a multiple comparison called Tukey's test was used to compare the difference between the mean values pair wise. Figure 13 displays the multiple comparisons for the 4 scenarios with 95% confidence interval. As the figure shows, these intervals have no overlap; therefore, the mean values are significantly different. Considering figure 13, we understand that scenarios 2 and 4 significantly show a better performance over scenarios 1 and 3.

As figure 14 shows, the performance of the fault prediction of the models based on the *training sampled data* (i.e. scenarios 2 and 4) is better than the *training original data* (i.e. scenario 1 and 3), regardless of selection of the features from the sampled or original data.

In addition, scenario 4 (i.e. training sampled data and selection of features from the sampled data) shows the *most* mean value of performance using all datasets. If we call the *impact* of a scenario as the mean value for the performance of all classifiers using all datasets, the deviation of scenario 2 (i.e. training the sampled data and selection of features from the original data) from the *impact mean* is the *best* (least).

Overall, scenario 2 achieves better than the others. Therefore, considering figure 14, we came into this conclusion that the *impact* of the scenarios on the performance of the classifiers is 3<1<2<4 if we consider the mean value, and 2<4<1<3 if we do the standard deviation value.

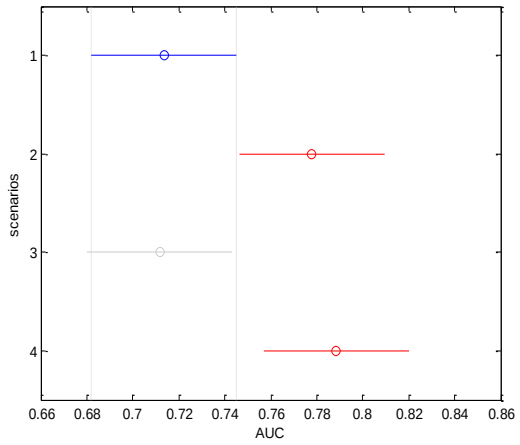

Figure 13. Multiple comparison for four scenarios.

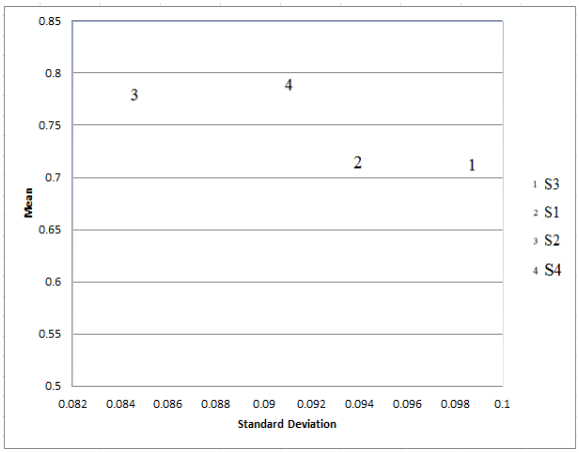

Figure 14. Mean value of performance of all classifiers (impact) for each scenario and deviation of scenarios from their impact using all datasets.

6. Conclusions and future work

This paper experimentally evaluated the ability of 17 classifiers in predicting fault-prone software modules with 4 scenarios in the context of 13 cleaned NASA datasets.

In a given classification problem, an important challenge is the choice of the convenient features when the underlying data is imbalanced. To deal with this problem, we discussed the different combinations of the feature selection and data sampling to create the training data for construction of a software fault prediction model. This study answered these research questions: (1) for which of the original or sampled data should feature selection be used? (2) given a set of picked features, based on which type of data (the original or sampled), we create the training data? (3) which of the classifiers have a better performance?

The results obtained showed that feature selection based on sampled or original data is not affected in the performance of the fault prediction models. Furthermore, the performance of the fault prediction models is better when the training data is created using the sampled data over the original data. In addition, *Bagging*, *Random Forest*, and *K** have the best performance in the mean for all datasets with scenarios 2 and 4.

A future work may conduct the additional experimental studies on the other datasets, feature selection, and sampling methods, and may use additional in dependent variables (i.e. features) such as *coupling* and *cohesion* metrics.

Another new idea that may be considered as the future work is thinking of the fault prediction of the concurrent programs such as the multi-thread programs. For such programs, metrics such as the number of concurrent and sequential threads should be considered. Two significant classes for such programs are (1) execution sequences of a concurrent program leading to deadlock, and (2) those not leading to deadlock. We have an experience on fault prediction of concurrent programs using the NARX neural network, where executions are classified into deadlock-prone and non-deadlock-prone [66]. However, this classification was based on the runtime (dynamic) behavior of the concurrent programs, and not the use of software static metrics.

Another future work may apply sample reduction to training phase inspired by [67]. In [67], authors addressed an instance reduction method to discard irrelevant instances from the training set.

References

[1] Menzies, T., Greenwald, J., & Frank, A. (2007). Data mining static code attributes to learn defect predictors. IEEE transactions on software engineering, vol. 33, no. 1, pp. 2-13.

[2] Catal, C., & Diri, B. (2009). Investigating the effect of dataset size, metrics sets, and feature selection techniques on software fault prediction problem.Information Sciences, vol. 179, no. 8, pp. 1040-1058.

[3] Jiang, Y., Lin, J., Cukic, B., & Menzies, T. (2009). Variance analysis in software fault prediction models. In the 20th International Symposium on Software Reliability Engineering, pp. 99-108.

[4] Khoshgoftaar, T. M., Rebours, P., & Seliya, N. (2009). Software quality analysis by combining multiple projects and learners. Software quality journal, vol. 17, no. 1, pp. 25-49.

[5] Lessmann, S., Baesens, B., Mues, C., & Pietsch, S. (2008). Benchmarking classification models for software defect prediction: A proposed framework and

novel findings. IEEE Transactions on Software Engineering, vol. 34, no. 4, pp. 485-496.

[6] Shepperd, M., Song, Q., Sun, Z., & Mair, C. (2013). Data quality: Some comments on the NASA software defect datasets. IEEE Transactions on Software Engineering, vol. 39, no. 9, pp. 1208-1215.

[7] Khoshgoftaar, T. M., Seliya, N., & Sundaresh, N. (2006). An empirical study of predicting software faults with case-based reasoning. Software Quality Journal, vol. 14, no. 2, pp. 85-111.

[8] Hall, M. A. (1999). Correlation-based feature selection for machine learning (Doctoral dissertation, The University of Waikato), www.cs.waikato.ac.nz/~mhall/thesis.pdf, Access date: 11/21/2016.

[9] Chawla, N. V., et al. (2011). SMOTE: synthetic minority over-sampling technique. Journal of Artificial Intelligence Research, vol. 16, pp. 321-357.

[10] Kamei, Y., Monden, A., Matsumoto, S., Kakimoto, T., & Matsumoto, K. I. (2007). The effects of over and under sampling on fault-prone module detection. In the 1st International Symposium on Empirical Software Engineering and Measurement, pp. 196-204.

[11] Riquelme, J. C., Ruiz, R., Rodríguez, D., & Moreno, J. (2008). Finding defective modules from highly unbalanced datasets. In the Workshops of the Conference on Software Engineering and Databases, vol. 2, no. 1, pp. 67-74.

[12] Catal, C., & Diri, B. (2007). Software defect prediction using artificial immune recognition system. In the 25th conference on IASTED International Multi-Conference: Software Engineering, pp. 285-290.

[13] Shirabad, J. S., & Menzies, T. J. (2005). The PROMISE repository of software engineering databases. http://promise.site.uottawa.ca/SERepository, Access date: 11/21/2016.

[14] Catal, C., & Diri, B. (2008). A fault prediction model with limited fault data to improve test process. In International Conference on Product Focused Software Process Improvement, pp. 244-257.

[15] Catal, C., & Diri, B. (2007). Software fault prediction with object-oriented metrics based artificial immune recognition system. In International Conference on Product Focused Software Process Improvement, pp. 300-314.

[16] Elish, K. O., & Elish, M. O. (2008). Predicting defect-prone software modules using support vector machines. Journal of Systems and Software, vol. 81, no. 5, pp. 649-660.

[17] Kanmani, S., Uthariaraj, V. R., Sankaranarayanan, V., & Thambidurai, P. (2007). Object-oriented software fault prediction using neural networks. Information and software technology, vol. 49, no. 5, pp. 483-492.

[18] Gondra, I. (2008). Applying machine learning to software fault-proneness prediction. Journal of Systems and Software, vol. 81, no. 2, pp. 186-195.

[19] Malhotra, R., & Singh, Y. (2011). On the applicability of machine learning techniques for object oriented software fault prediction. Software Engineering: An International Journal, vol. 1, no. 1, pp. 24-37.

[20] Liu, H., & Yu, L. (2005). Toward integrating feature selection algorithms for classification and clustering. IEEE Transactions on knowledge and data engineering, vol. 17, no. 4, pp. 491-502.

[21] Hall, M. A., & Holmes, G. (2003). Benchmarking attribute selection techniques for discrete class data mining. IEEE transactions on knowledge and data engineering, vol. 15, no. 6, pp. 1437-1447.

[22] Saeys, Y., Abeel, T., & Van de Peer, Y. (2008). Robust feature selection using ensemble feature selection techniques. In Joint European Conference on Machine Learning and Knowledge Discovery in Databases, pp. 313-325.

[23] Khoshgoftaar, T. M., Gao, K., & Napolitano, A. (2014). Improving software quality estimation by combining feature selection strategies with sampled ensemble learning. In the 15th IEEE International Conference on Information Reuse and Integration, pp. 428-433.

[24] Gao, K., & Khoshgoftaar, T. M. (2011). Software Defect Prediction for High-Dimensional and Class-Imbalanced Data. In the 23rd International Conference on Software Engineering & Knowledge Engineering, pp. 89-94.

[25] Gao, K., Khoshgoftaar, T. M., & Napolitano, A. (2015). Combining feature subset selection and data sampling for coping with highly imbalanced software data. In the 27th International Conference on Software Engineering and Knowledge Engineering, pp. 439-444.

[26] Gao, K., Khoshgoftaar, T. M., & Wald, R. (2014). Combining Feature Selection and Ensemble Learning for Software Quality Estimation. In the 27th International Florida Artificial Intelligence Research Society Conference.

[27] Wang, H., Khoshgoftaar, T. M., Gao, K., & Seliya, N. (2009). Mining data from multiple software development projects. In IEEE International Conference on Data Mining Workshops, pp. 551-557.

[28] Engen, V., Vincent, J., & Phalp, K. (2008). Enhancing network based intrusion detection for imbalanced data. International Journal of Knowledge-Based and Intelligent Engineering Systems, vol. 12, no. 5- 6, pp. 357-367.

[29] Kamal, A. H., Zhu, X., Pandya, A. S., Hsu, S., & Shoaib, M. (2009). The impact of gene selection on

imbalanced microarray expression data. In the 1st International Conference on Bioinformatics and Computational Biology, pp. 259-269.

[30] Zhao, X. M., Li, X., Chen, L., & Aihara, K. (2008). Protein classification with imbalanced data. Proteins: Structure, function, and bioinformatics, vol. 70, no. 4, pp. 1125-1132.

[31] Chawla, N. V., Bowyer, K. W., Hall, L. O., & Kegelmeyer, W. P. (2002). SMOTE: synthetic minority over-sampling technique. Journal of Artificial Intelligence Research, vol. 16, pp. 321-357.

[32] Cieslak, D. A., Chawla, N. V., & Striegel, A. (2006). Combating imbalance in network intrusion datasets. In the IEEE International Conference on Granular Computing, pp. 732-737.

[33] Seiffert, C., Khoshgoftaar, T. M., & Van Hulse, J. (2009). Improving software-quality predictions with data sampling and boosting. IEEE Transactions on Systems, Man, and Cybernetics-Part A: Systems and Humans,vol. 39, no. 6, pp. 1283-1294.

[34] Chen, Z., Menzies, T., Port, D., & Boehm, D. (2005). Finding the right data for software cost modeling. IEEE software,vol. 22, no. 6, pp. 38-46.

[35] Liu, H., Motoda, H., & Yu, L. (2004). A selective sampling approach to active feature selection. Artificial Intelligence, vol. 159, no. 1, pp. 49-74.

[36] Khoshgoftaar, T. M., Gao, K., & Seliya, N. (2010). Attribute selection and imbalanced data: Problems in software defect prediction. In 22nd IEEE International Conference on Tools with Artificial Intelligence, vol. 1, pp. 137-144.

[37] Dietterich, T. G. (1998). Approximate statistical tests for comparing supervised classification learning algorithms. Neural Computation, vol. 10, no.7, pp. 1895-1923.

[38] Duda, R. O., Hart, P. E., & Stork, D. G. (2012). Pattern classification. John Wiley & Sons.

[39] Kothari, C. R. (2004). Research methodology: Methods and techniques. New Age International.

[40] Le Cessie, S., & Van Houwelingen, J. C. (1992). Ridge estimators in logistic regression. Applied statistics, pp.191-201.

[41] Breiman, L. (2001). Random forests. Machine learning, vol. 45, no. 1, pp. 5-32.

[42] Biau, G., Devroye, L., & Lugosi, G. (2008). Consistency of random forests and other averaging classifiers. Journal of Machine Learning Research, vol. 9, pp. 2015-2033.

[43] Fenton, N. E., & Ohlsson, N. (2000). Quantitative analysis of faults and failures in a complex software system. IEEE Transactions on Software engineering, vol. 26, no. 8, pp. 797-814.

[44] Livingston, F. (2005). Implementation of Breiman's random forest machine learning algorithm. Machine Learning : ECE519.

[45] Freund, Y., & Schapire, R. E. (1996). Experiments with a new boosting algorithm. In the 13th International Conference on Machine Learning, pp. 148-156. 1996.

[46] Friedman, J., Hastie, T., & Tibshirani, R. (2000). Additive logistic regression: a statistical view of boosting (with discussion and a rejoinder by the authors).The Annals of Statistics, vol. 28, no. 2, pp. 337-407.

[47] Witten, I. H., & Frank, E. (2005). Data Mining: Practical machine learning tools and techniques. Morgan Kaufmann.

[48] Haykin, S. S. (2001). Neural networks: a comprehensive foundation. Tsinghua University Press.

[49] Broomhead, D.S. & Lowe, D. (1988). Radial basis functions, multi-variable functional interpolation and adaptive networks. Royal Signals and Radar Establishment Publisher.

[50] Park, J., & Sandberg, I. W. (1991). Universal approximation using radial-basis-function networks. Neural Computation, vol. 3, no. 2, pp. 246-257.

[51] John, G. H., & Langley, P. (1995). Estimating continuous distributions in Bayesian classifiers. In the 11th Conference on Uncertainty in artificial intelligence, pp. 338-345.

[52] Heckerman, D. (1998). A tutorial on learning with Bayesian networks. Learning in graphical models, MIT Press Cambridge, pp. 301-354.

[53] Vapnik, V. (2000). The nature of statistical learning theory. 2nd Edition, Springer.

[54] Press, W. H., Teukolsky, S. A., Vetterling, W. T., & Flannery, B. P. (2007). Numerical Recipes: The Art of Scientific Computing, Section 16.5, Support Vector Machines, Cambridge University Press, The 3rd Edition,

[55] Cleary, J. G., & Trigg, L. E. (1995). K*: An instance-based learner using an entropic distance measure. In the 12th International Conference on Machine learning, vol. 5, pp. 108-114.

[56] Iba, W., & Langley, P. (1992). Induction of one-level decision trees. In the 9th International Conference on Machine Learning, pp. 233-240.

[57] Quinlan, J. R. (2014). C4.5: programs for machine learning, Morgan Kaufmann Publishers.

[58] Freund, Y., & Mason, L. (1999). The alternating decision tree learning algorithm. In the 16th International Conference on Machine Learning, vol. 99, pp. 124-133.

[59] Holmes, G., Pfahringer, B., Kirkby, R., Frank, E., & Hall, M. (2002). Multiclass alternating decision

trees. In the European Conference on Machine Learning, pp. 161-172.

[60] Frank, E., & Witten, I. H. (1998). Generating accurate rule sets without global optimization. In the 15th International Conference on Machine Learning, pp. 144-151.

[61] McCabe, T. J. (1976). A complexity measure. IEEE Transactions on Software Engineering, vol. 2, no. 4, pp. 308-320.

[62] McCabe, T. J., & Butler, C. W. (1989). Design complexity measurement and testing. Communications of the ACM, vol. 32, no.12, pp. 1415-1425.

[63] Kohavi, R. (1995). A study of cross-validation and bootstrap for accuracy estimation and model selection. In the 14th International Joint Conference on Artificial Intelligence, vol. 2, pp. 1137-1143.

[64] Rutherford, A. (2011). ANOVA and ANCOVA: a GLM approach. John Wiley & Sons.

[65] One-way analysis of variance-MATLAB anova1-MathWorks (2016), http://www.mathworks .com/help/stats/anova1.html, Access date: 11/21/2016.

[66]. Babamir, S. M., Hassanzade, E., & Azimpour, M. (2015). Predicting potential deadlocks in multithreaded programs. Concurrency and Computation: Practice and Experience, vol. 27, no.17, pp. 5261-5287.

[67] Hamidzadeh, J. (2015). IRDDS: Instance reduction based on distance-based decision surface, Journal of IA and Data Mining, vol. 3, no. 2, pp. 121-130.

نشریه هوش مصنوعی و داده کاوی

ارزیابی دسته‌بندها در پیش‌بینی پیمانه‌های مستعدخطای نرم‌افزار

سید مرتضی بابامیر* و فاطمه کریمیان

دانشکده برق و کامپیوتر، دانشگاه کاشان، کاشان، ایران.

ارسال ۲۰۱۶/۰۴/۲۹؛ بازنگری ۲۰۱۶/۰۶/۱۲؛ پذیرش ۲۰۱۶/۱۰/۳۰

چکیده:

قابلیت اطمینان نرم‌افزار به تعداد پیمانه‌های مستعد-خطای آن بستگی دارد. یعنی هرقدر پیمانه‌های مستعد خطای نرم‌افزار کمتر باشند، اعتماد به نرم-افزار بیشتر می‌شود. بنابراین اگر قادر به پیش‌بینی تعداد پیمانه‌های مستعد-خطای نرم افزار باشیم، قضاوت درباره قابلیت اطمینان نرم‌افزار نیز امکان پذیر خواهد بود. در پیش‌بینی پیمانه‌های مستعد-خطای نرم‌افزار، یکی از ویژگی‌های کمکی، معیار نرم‌افزار است که توسط آن می‌توان پیمانه‌های نرم‌افزار را به مستعد-خطا و بدون-مستعد -خطا دسته‌بندی کرد. برای ایجاد این دسته‌بندی، بر روی ۱۷روش دسته‌بند تحقیق کردیم که ویژگی‌هایشان ۱۹ معیار نرم‌افزار و نمونه‌هایشان(پیمانه‌های نرم‌افزار) ۱۳ مجموعه داده‌ی ناسا هستند.

دو موضوع مهم که بر روی صحت پیش‌بینی در هنگام استفاده از روش‌های داده‌کاوی اثر می‌گذارد عبارتند از: (۱) انتخاب بهترین ویژگی‌ها (مانند معیارهای نرم‌افزار) در میان تنوع وسیع ویژگی‌ها و (۲) نمونه‌برداری از موارد به منظور متوازن کردن موردهای ناهماهنگ؛ هنگامی‌که دسته‌بند به سمت کلاس اکثریت متمایل می‌شود دو کلاس ناهماهنگ داریم. بر اساس انتخاب ویژگی و نمونه‌برداری از مورد، ۴ سناریو برای ارزیابی ۱۷ روش دسته‌بند به‌منظور پیش‌بینی پیمانه‌های مستعد-خطا درنظر گرفتیم. برای انتخاب ویژگی‌ها، از انتخاب ویژگی مبتنی بر همبستگی و برای نمونه‌برداری موردها روش نمونه‌برداری بیش از حد اقلیت ترکیبی را استفاده کردیم. نتایج تجربی نشان دادند که نمونه‌برداری مناسب پیمانه‌ها، تاثیر قابل توجهی بر روی صحت پیش‌بینی قابلیت اطمینان نرم‌افزار می‌گذارند، اما انتخاب معیار اثر قابل توجهی بر روی پیش‌بینی ندارد. همچنین در میان دسته‌بندها، Bagging ، K^* و جنگل تصادفی هنگامی‌که موردهای نمونه‌برداری شده را برای داده‌های آموزشی استفاده می‌کنیم بهترین دسته‌بندها بودند.

کلمات کلیدی: پیش‌بینی خطای نرم‌افزار، کارایی دسته‌بند، انتخاب ویژگی، نمونه‌برداری از داده، معیار نرم‌افزار.