

# A dimensionality reduction-based efficient software fault prediction using Fisher linear discriminant analysis (FLDA)

Anum Kalsoom<sup>1</sup> · Muazzam Maqsood<sup>1,2</sup> ·  
Mustansar Ali Ghazanfar<sup>2</sup> · Farhan Aadil<sup>1,2</sup> ·  
Seungmin Rho<sup>3</sup> 

Published online: 20 March 2018

© Springer Science+Business Media, LLC, part of Springer Nature 2018

**Abstract** Software quality is an important factor in the success of software companies. Traditional software quality assurance techniques face some serious limitations especially in terms of time and budget. This leads to increase in the use of machine learning classification techniques to predict software faults. Software fault prediction can help developers to uncover software problems in early stages of software life cycle. The extent to which these techniques can be generalized to different sizes of software, class imbalance problem, and identification of discriminative software metrics are the most critical challenges. In this paper, we have analyzed the performance of nine widely used machine learning classifiers—Bayes Net, NB, artificial neural network, support vector machines, K nearest neighbors, AdaBoost, Bagging, Zero R, and Random Forest for software fault prediction. Two standard sampling techniques—SMOTE and Resample with substitution are used to handle the class imbalance problem. We further used FLDA-based feature selection approach in combination with SMOTE and Resample to select most discriminative metrics. Then the top four classifiers based on performance are used for software fault prediction. The experimentation is carried out over 15 publically available datasets (small, medium and large) which are collected from PROMISE repository. The proposed Resample-FLDA method gives better performance as compared to existing methods in terms of precision, recall,  $f$ -measure and area under the curve.

---

✉ Seungmin Rho  
smrho@sungkyul.ac.kr

<sup>1</sup> Department of Computer Science, COMSATS Institute of Information and Technology Attock, Attock, Pakistan

<sup>2</sup> Department of Software Engineering, University of Engineering and Technology Taxila, Taxila, Pakistan

<sup>3</sup> Department of Media Software, Sungkyul University, Anyang, South Korea

**Keywords** Software fault prediction · Fisher linear discriminant · Reliability · Fault-tolerance · Robustness

## 1 Introduction

The software industry has seen enormous growth due to its high use in daily life. The size and, ultimately, the complexity of software modules are rapidly increasing these days. This increase is giving rise to customer demands as well as for the software to be reliable and secure. It is practically impossible to create error-free and reliable software due to budget and time constraints. In the life cycle of software, the strategy adopted for dealing software faults must be properly planned. These faults, which, in case, are not removed, cause quality failure and cost escalation. Software Quality Assurance (SQA) is an important process to achieve the required software quality at a minimal cost. Different SQA processes like formal code inspections, code walkthroughs, software testing and software fault prediction are more likely to be included in it. Software fault prediction has now become a mandatory step in the life cycle of software to identify fault-prone modules in early stages of software development before testing time [1, 2].

The objective of software fault prediction is to identify the software faults before testing phase by using certain software attributes or metrics. Software fault prediction models are constructed using previous releases of similar projects and perform fault prediction for current software. Software fault prediction provides assistance in allocating SQA resources in an economical and efficient way by predicting faulty components of software. Identification of fault-prone software modules at early phases of software development helps to improve the quality of software system [3]. This enables us to guide software testers to bring into focus fault-prone components in the first place. This can only be achieved by the identification of software quality metrics such as fault percentage, required effort, testability, maintainability and reliability at the initial developmental phase. Various software fault prediction approaches are proposed for the prediction of fault-prone and non-fault-prone modules using Halstead and McCabe metrics [1].

Various software metrics and approaches are available for fault prediction [4]. Number of approaches have been proposed previously for software fault detection, many of which aim to classify software modules into faulty or non-faulty categories. The aim is to utilize the success of machine learning classification techniques which include ANN, SVMs, Linear Regression (LR), Decision Tree, NB, Genetic Programming and Random Forest (RF) for software fault detection. Some researchers have used ensemble methods for classification of faulty modules. In most cases, the results for these classifiers cannot be generalized for large-sized software. Performance of these above-mentioned classifiers is severely affected by data quality; biased datasets [5] curse of dimensionality [6] and class imbalance problem [7]. The dimensionality problem is caused by a lot of unnecessary features (i.e., software metrics) that can be solved by feature selection. The class imbalance problem is caused by redundant instances of similar classes and is handled by instance sampling (or reduction) which, from majority classes, samples a subset of the instances. Although both techniques proved them

to be effective solutions, few researchers have merged feature selection and instance reduction simultaneously. This has been done for the sake of improving data quality in software fault prediction [8, 9].

The datasets used for software fault prediction comprises of following class categories: “fault-prone” and “non-fault-prone.” Generally, the fault-prone category is under-represented in data sets. Therefore, development of an efficient software quality models is necessary for the identification of change prone classes. It is because all of these classes must be predicted with ultimate efficiency so that software’s quality can be improved. A classification model which can be labeled as efficient and effective must be able to detect classes of both kinds of nature which are “fault-prone” and “non-fault-prone,” that too with high precision and accuracy. However, what generally happens is that non-fault-prone classes are predicted with high accuracy, while fault-prone classes are predicted with less accuracy. This problem is generally caused by the imbalanced nature of training data. Since the datasets are influentially imbalanced, this lower recognition in case of fault-prone classes can be easily ignored and better accuracy rates are showcased by classifiers on the whole. However, this case can be unfavorable and may lead to inexact judgments causing losses and poor reputation of software organization. For fault-prone classes, such low prediction accuracy is not advisable as it would lead to a bad quality software product. This is so because there is a requirement of allocating sufficient resources to fault-prone classes so that they can be accurately constructed, scrutinized, demonstrated and tested [10]. These classes must be properly handled to prevent occurrences of any kind of faults in software’s future version. Keeping these issues in mind, the objectives of this research paper are (1) to study the effect of software fault prediction dataset size on the performance of most widely used machine learning classifiers (2) to handle the class imbalance problem for software fault prediction and (3) to use an effective approach for efficient feature selection and dimensionality reduction using Fisher linear discriminant analysis (FLDA).

In this paper, the aforementioned problems are addressed by proposing an efficient algorithm that can handle dimensionality reduction and class imbalance problem. We have explored the effect of data sampling techniques and a dimensionality reduction technique on the performance of machine learning classifiers. To start with, most widely used machine learning classifiers are used on a large number of publically available small-, medium- and large-size datasets. Two standard sampling techniques named SMOTE and Resample, with replacement, are used to handle the class imbalance problem [11]. After using sampling methods, the behavior of various machine learning approaches clearly progressed toward betterment. FLDA is used separately for the selection of most discriminative software metrics and dimensionality reduction. We designed a new approach to handle dimensionality reduction and feature selection issue by using FLDA. It is a supervised learning dimensionality reduction approach that selects the most discriminative features with respect to class labels. We intend to utilize this functionality to improve performance. To the best of our knowledge, FLDA is not used for software fault detection so far. Using FLDA significantly improves the performance of machine learning classifiers for datasets of all sizes. Machine learning classifiers produced exceptional results when Resample with replacement and FLDA is used together. This paper has the following contributions;

- We performed an analysis to check the performance of most widely used machine learning classifiers on software fault prediction datasets of different sizes.
- An FLDA-based dimensionality reduction approach is proposed to detect most discriminative software metrics for software fault prediction.
- We explored the performance of two state-of-the-art sampling techniques in combination with FLDA. The results have been significant.
- To verify generalizability of proposed method by evaluating it over 15 publically available datasets including small-, medium- and large-sized software.
- The proposed method not only correctly identifies non-faulty modules but also correctly classify faulty modules.

The rest of the paper is organized as follows; Sect. 2 represents related work of classification, feature selection and class imbalance issue for software fault prediction. Section 3 explains the methodology for software fault prediction. Section 4 presents experimental methodology and Sect. 5 explains results followed by the conclusion.

## 2 Related work

A lot of work is done in the field of software fault prediction. Related work is divided into three sections according to the objectives of this paper. The first section covers the literature related to software fault prediction. The second section explains work done in the area of class imbalance problem followed by related work of feature selection for software fault prediction.

### 2.1 Software fault prediction

In relation to fault prediction techniques, certain studies are reportedly using generalized linear regression, Poisson regression, negative binomial regression, genetic programming, and neural network.

Graves et al. [12] performed different experiments to predict a number of faults using generalized linear regression (GLR). These experiments were performed for a large telecommunication company using various software change history metrics. They proposed two kinds of different models for a number of fault prediction. In the first experiment, a stable model was designed using many “past faults” for fault prediction. After this stable model, another model was presented that uses certain change history metrics and it was called a GLR-based model. Comparison of these two models showed that GLR-based model performed better than the stable model. However, the combination of “Module age,” “Modular changes” and “Lifespan of changes” metrics produced better results for software fault prediction. Other software metrics like module size and its complexity performed poorly for fault prediction. The primary aim of this study was to evaluate the performance of different change history metrics and their combinations for software fault prediction using GLR. However, other machine learning prediction algorithms were not explored in this study.

Another study presented by critical analysis is performed for the use of negative binomial regression to predict fault density and number of faults [13]. The experiments

were performed on two industry projects using Lines of Codes (LOC) and different file characteristics. The results suggested that negative binomial regression performed well for software fault prediction [14]. However, later on, another fault prediction model was designed, based on LOC metric, which produced comparable results to negative binomial regression (NBR). This not only requires less effort to design software fault prediction model but also generate accurate results. Evaluation of results was based on performance; evaluation none other than faults found in the top 20% file predicted to be fault-prone [13, 15]. Some other studies have also been reported by Janes et al. [16] using NBR. Janes et al. designed an NBR model [15] for telecommunication system using object-oriented metrics to predict the fault counts. It was claimed that NBR produced comparatively better results to predict fault counts. Yu [14] has performed a comparative analysis of NBR and logistic regression for fault prediction. Logistic regression performed comparatively better for the prediction of fault-prone software modules and NBR performed better for prediction of multiple faults in a software module.

Ensemble methods have been largely used for software fault prediction in recent years, especially for binary classification. A similar ensemble method was presented by Misirli et al. [17] for software fault prediction using a combination of three different techniques. These techniques were Naïve Bayes, ANN, and Voting Feature Intervals. The results suggested that an ensemble classifier produced significantly better prediction accuracy in comparison with Naïve Bayes classifier alone. In another study performed by Zheng [18], a comparative analysis of three cost-effective boosting neural networks for software fault prediction was presented. The results claimed that cost-sensitive neural networks achieved significantly accurate prediction for software defect prediction. Twala [19] assessed ensemble classification methods to predict faults for a large space system. In this study, five fault prediction approaches were used as base learners for ensemble method. The result suggested that ensemble methods improved prediction accuracy every time in comparison with individual classifiers.

Wang et al. [20] presented a comparative analysis of various ensemble methods with Naïve Bayes for software defect prediction. According to results, ensemble methods—RF and voting—produced better prediction accuracy. Other studies like [4, 21] have proposed ensemble methods based work for fault prediction and have also compared performances of ensemble method with other fault prediction techniques. Aljaman and Alish [4] used bagging and boosting for software fault prediction and produced a better performance as compared to individual classifiers used for software fault prediction. A study was conducted by Khoshgoftaar et al. [21] in the year 2003 for performance evaluation of three combination techniques using ensemble method in predicting software quality. Results suggested that combination approaches proved to generate more efficient performance for prediction. Certain ensemble methods were recently investigated for the sake of software prediction maintenance/changing efforts [4]. This study was planned and evaluated over two publicly available datasets using some design level software metrics. The ensemble method proved to give better prediction accuracy results as compared to other individual classifiers.

## 2.2 Class imbalance problem in software fault prediction

It is mandatory to handle the class imbalance problem for efficient model development. A brief overview regarding nature and issues associated with the field of imbalanced learning [4]. According to them, these primary causes, accuracy, class distribution and error cost lead to the ineffective performance of certain learning methods. Class imbalance can be handled using some methods and these broad categories suggested to handle class imbalance as follows: (1) incorporating the use of sampling method, (2) use of cost-sensitive methods, (3) active learning and kernel-based methods, (4) use of ensemble learners, (5) application of some specific evaluation metrics, (6) incorporation of human knowledge, (7) segmentation of data, (8) non-greedy methods for used for searching, (9) use of an effective inductive bias and (10) other methods like unary classification methods or novelty detection methods. It is worth mentioning that some classification methods do not assume the imbalanced nature of data. It has been widely useful in the development of classification models particularly for the imbalanced dataset in the field of quality engineering. However, these methods are rarely used in software engineering because of difficulty in determining a suitable threshold for the execution of an efficient classification process.

In recent years, a large number of applications face class imbalance problem especially sentiment analysis, fraud detection, video mining, text mining, churn prediction and other bioinformatics applications. Researchers have explored various methods to handle class imbalance problem for software defect prediction. Wang and Yao [22] conducted a study using threshold mining, resampling, and ensembles to predict software defects by using imbalanced datasets [23]. Seiffert et al. [24] to explore sampling methods to improve the performance of software fault prediction models. Seliya and Khoshgoftaar [6] explored the cost-sensitive learning techniques using decision trees to develop software defect prediction models. The misclassification cost was taken as the key parameter for model training. Galar et al. [25] and Rodriguez et al. [26] presented a comparison of cost-sensitive, sampling and ensemble learning methods development for software defect prediction using imbalance data. Gao et al. [6] suggested that use of feature selection with sampling techniques improves the performance of software fault prediction.

## 2.3 Feature selection approaches

In this section, we discuss the importance of feature selection in the background of software fault prediction. We then discuss previous related work about feature selection and dimensionality reduction. Software fault prediction is an important step for getting to know about faulty software modules. Many researchers prefer machine learning classification models to predict these faulty modules. These classification models require training data collected from previous projects where faulty modules have been identified. It is widely discussed that dataset quality plays an important role to increase prediction accuracy of a classification algorithm. The performance of these machine learning algorithms can be further improved by data preprocessing that includes feature selection and instance reduction. Feature selection process consists of

identifying and discarding irrelevant features from a dataset so that only discriminant features are selected for training classification models. Some feature selection methods have been widely used that are categorized as *filter-based* and *wrapper-based*. Filter-based methods select features that are most relevant based on the correlation between features and class labels. Wrapper-based methods require feedback from classification model and select feature vector iteratively that may lead to high computational complexity. Researchers have conducted a comparison between filter- and wrapper-based feature selection for software fault prediction). Shivaji et al. [26] evaluated five feature selection methods including three filter-based ranking methods and two wrapper-based methods using Naïve Bayes and SVM. All the feature selection methods proved to improve the performance of software fault prediction. However, the improvement was only comparable for classifiers after using feature selection methods.

Gao et al. [6] performed a comparison of feature selection methods in predicting faulty software modules for a large legacy telecommunication system. They have used seven filter-based methods and three wrapper-based feature selection methods using search-based greedy techniques. This result suggested that removing 85% of software metrics did not affect prediction accuracy and even the performance was improved in some cases. Wang et al. [22] presented a comparative analysis for evaluation of seventeen ensembles of eighteen feature ranking methods. The results suggested that use of few rankers (i.e., 2–4) improved the results. Dimensionality reduction is a process to select a subset of representative features (software metrics) to train a classification model [11]. Researchers have used different dimensionality reduction techniques for software fault prediction recently. Dimensionality reduction removes redundant features to handle the problem of inter-class imbalance. Random sampling is one of the effective methods for instance reduction which is also effective in minimizing the impact of imbalanced distributions among classes [6].

Experimental results suggested that a combination of random under-sampling and Naïve Bayes yielded good performance for highly imbalanced data. Pelayo et al. [35] presented a comparative analysis between random under-sampling and oversampling for six software datasets. The statistical analysis suggested that under-sampling proved useful in improving prediction performance of classification algorithms. Khoshgoftaar et al. [6] discussed the effects of random sampling combined with other data preprocessing methods including feature ranking. Their results also suggested the effectiveness of random sampling to deal imbalanced datasets. Until recently, only a few researchers have combined feature selection with sampling to handle data preprocessing for software fault prediction. Liu et al. [37] combined feature selection methods with instance sampling for software fault prediction. However, the purpose of instance sampling was to reduce a total number of instances instead of handling class imbalance.

In this study, we have handled all aforementioned issues related to software fault detection. The suitability of different machine learning classifiers are explored for large-sized datasets and best four classifiers are selected based on the performance. SMOTE and Resample methods are used to handle class imbalance issue. We incorporated FLDA with these sampling issues to handle feature selection problem.

### 3 Proposed methodology

The proposed mythology for software fault prediction using preprocessing and FLDA is outlined in Fig. 1. Each step is explained in the following sections.

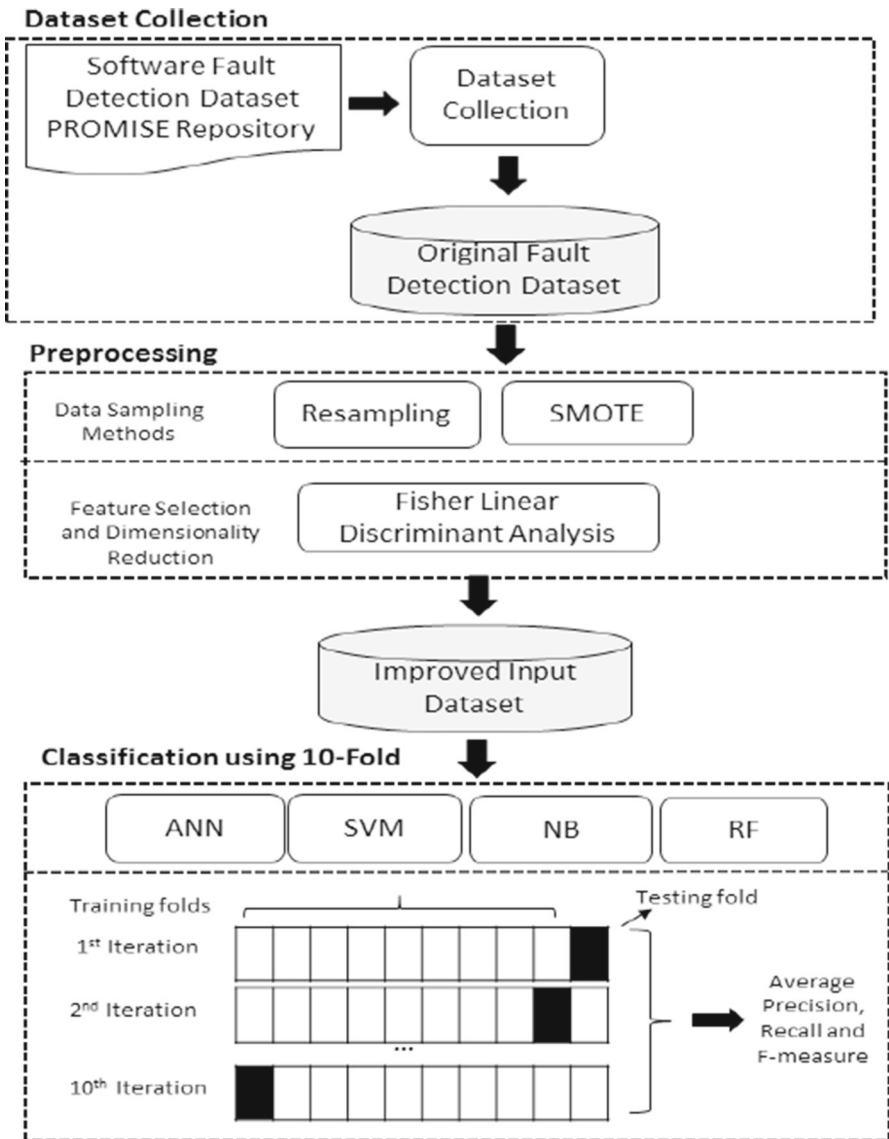


Fig. 1 Proposed methodology for software fault prediction



**Table 1** Details of classifiers used for software fault detection

Classifiers	Description
Bayes.Net	Bayesian uses interdependence of various attributes and Bayes theorem
Naïve Bayes	This technique uses Bayes theorem and also assumes no interdependency between features
MLP	MLP uses two separate passes; backward and forward propagation
SMO	This technique uses hyperplane to separate the data points
IBK(KNN)	This uses nearest neighbors to assign class labels to new objects
Adaboost M1	It is a meta learning approach for classification that uses one level decision tree
Bagging	It uses meta learning and voting technique in order to classify data efficiently
Zero R	Zero R is the simplest classification method which relies on target and ignores all predictors
Random Forest	It is an ensemble technique that uses multiple trees to detect fault detection

### 3.1 Fault prediction techniques

This section presents machine learning classifiers used in this study. As one of the prime objectives of this study is to explore the suitability of different machine classifiers for software fault prediction, therefore, nine most widely used machine learning classifiers have been explored for software fault detection. These classifiers include Bayesian, Naïve Bayes (NB), ANN, SVMs, KNN, AdaBoost, Bagging, Zero R, and RF. The details of these classifiers are presented in Table 1. Best four classifiers based on performance are further picked and used for evaluation of the proposed system. The design details for these four algorithms are given in this section.

#### 3.1.1 Support vector machines (SVM)

In general, SVM produces excellent results for balanced datasets, but it is sensitive to imbalanced datasets and produces sub-optimal results. It has been observed that separating hyperplane of SVM produced from an imbalanced data is often skewed toward minority class. This, ultimately, produces sub-optimal results with respect to a minority class. The SVM focuses to maximize margin while tries to minimize penalty term associated with misclassification. The same value of cost,  $C$ , is used for both classes. A number of misclassifications should be reduced in order to bring penalty term down. For an imbalanced dataset, the density of majority class is higher than the density of minority class, even around the separating hyperplanes. In ideal cases, hyperplane should pass through the classes. It has been observed that minority class objects are placed farther from the separating hyperplane due to their less quantity.

Therefore, hyperplane can be shifted (skewed) toward minority class to handle misclassifications. This shift toward minority class increases the possibility of false negative predictions. For extreme class imbalance cases, SVM can produce a high number of false negatives. SVM is composed of related supervised learning methods, holding a special property of concurrently minimizing the empirical classification error and maximizing the geometric margin. For this reason, it is also named as maximum margin classifier.

SVM can easily deal with high dimensional input space and it is appropriate for problems having sparse instances. If a binary classification is considered to be a problem of linearly separable classification, there can be a lot of decision boundaries. Decision boundary must be far away from the data of two classes in SVM and its training focuses on detection of the hyperplane between positive and negative training examples. Binary class SVM classifies a new vector  $d$  into a class by this rule:

$$\sum_{j=1}^{nsv} \alpha_j y_j d_j d'_j + b, \tag{1}$$

Here,  $nsv$  is the total number of support vectors. The factor  $\alpha_j$  describes support vectors which indicate decision boundary,  $y_i \in \{+1, -1\}$  shows class labels and  $d_j$  indicates training vectors. According to this,  $d'$  is being classified as class +1 if the sum is positive otherwise -1. SVM also deals with nonlinear decision boundary by the use of kernel trick [27]. The objective is to do the transformation of obtained input space into a higher dimensionality feature space. Once this transformation takes place, the linear operation becomes equal to nonlinear operation in input space of feature space. Complexity gets reduced, initiating ease of classification. This transformation is described as:

$$\emptyset : X \rightarrow F$$

Here,  $X$  and  $F$  denote input space and feature space, respectively. Example of polynomial kernel transformation is:

$$\emptyset(x_1, x_2) \rightarrow (x_1^2, x_2^2, \sqrt{x_1 x_2}, x_1, x_2, 1)$$

Once the transformation is done, above equation can be written as under:

$$\sum_{j=1}^n \alpha_j y_j \emptyset^T(d_j) \emptyset(d') + b \tag{2}$$

where  $K(d_j, d') = \emptyset^T(d_j) \emptyset(d')$  is a symmetric positive function named as kernel function of two variables. Other kernel functions which can be used include the linear kernel, polynomial kernel and radial basis kernel [33].

### 3.1.2 Random forest

RF is one of the most popular machine learning algorithms used for classification. RF is popular due to its good performance as compared to other classifiers. It is an ensemble technique that (grows many weak classifiers, i.e., decision trees) and classifies the instances by aggregating their votes. RF takes advantage of two powerful techniques for classification; Bagging and random feature selection. In bagging, trees are built using the bootstrap samples taken from training data. A top-down induction procedure is followed to favor the diversity of ensemble process for each random tree. Then prediction is made using majority vote. To design each tree, a subset of original features are considered,  $d \ll D$  where  $d$  is a subset of a complete feature vector with length  $D$ . Later on, RF randomly split these features at each node while building a tree. Each tree is built to its full depth and there is no pruning procedure used once the tree is built. Then finally, classification is done by considering majority votes.

### 3.1.3 Multi-layer perceptron

ANN or neural network is machine learning classifier inspired by human nervous system. ANN is formed by interconnecting many groups of artificially created neurons. These interconnected artificial neurons use connectionist approach for information computing. The ANN is an adaptive classification algorithm which adapts to the information that passes through network internally or externally. These ANN networks are a collection of parallel elements called nodes. The training for each ANN is done by adjusting connections between nodes. The network is trained by adjusting weights based on the difference between output and target class. Training stops when the difference between output and target class is minimum or Zero. Minimum difference between output and target implies that output matches the input. ANN is a supervised learning classifier and it is trained by adjusting weights for each node. The training process continues until there is a difference between output and target. It is sent back for readjustment of weights until the desired output is achieved. ANN is widely used in pattern recognition due to its knowledge storing ability. ANNs are categorized into single layer perceptron and Multi-Layer Perceptron (MLP). Single layer perceptron uses single layer of weights which means input is directly connected to the output. On the other hand, MLP uses multiple layers that are input layer, hidden layers, and output layers. This proposed work used an MLP based ANN using backpropagation algorithm for training.

### 3.1.4 Naïve Bayes

Based independently on Bayes theorem, the Naïve Bayes classifier is appropriate for high input dimensions. By using Bayes theorem, the probability of document  $d$ , which is apparently to be in class  $C_j$ , is calculated as under:

$$P(C_j|d) = \frac{P(C_j)P(d|C_j)P(d)}{P(d)} \quad (3)$$

Here  $P(C_j|d)$ ,  $P(C_j)$ ,  $P(d|C_j)$  and  $P(d)$  are named to be posterior, prior, likelihood, and evidence accordingly.

In Naïve Bayes classifier, we assume that features are independent of conditions. Here, considering an example, suppose that in a document, words (features) that appear are independent of one another [75]. However, if there is a set of features  $\omega_1, \dots, \omega_h$ , on that condition, we write numerator of Eq. (3) this way:

$$\begin{aligned} P(C_j)P(d|C_j) &= P(C_j)P(\omega_1, \dots, \omega_h|C_j) \\ &= P(C_j)P(\omega_1|C_j)P(\omega_2, \dots, \omega_h|C_j) \\ &= P(C_j)P(\omega_1|C_j) \dots P(\omega_h|C_j, \omega_1 \dots, \omega_{h-1}) \end{aligned} \tag{4}$$

According to naïve assumption, every feature  $\omega_i$  is do not conditionally depend on another feature  $\omega_j$  for

$$j \neq i, i.e P(\omega_i|C_j, \omega_j) = P(\omega_i|C_j)$$

We can simplify Eq. (4) this way:

$$\begin{aligned} P(C_j)P(d|C_j) &= P(C_j)P(\omega_1|C_j) \dots P(\omega_h|C_j, \omega_1 \dots, \omega_{h-1}) \\ &= P(C_j)P(\omega_1|C_j)P(\omega_2|C_j) \dots \end{aligned} \tag{5}$$

$$P(C_j) \prod_{x=1}^h P(\omega_x|C_j)$$

Substituting Eq. (5) into Eq. (3), we get

$$P(C_j|d) = \frac{P(C_j) \prod_{x=1}^h P(\omega_x|C_j)}{P(\omega_1, \dots, \omega_h)} \tag{6}$$

$$P(C_j|i) = \frac{P(C_j) \prod_{t=1}^H \prod_{x=1}^{|dx|} P(\omega_{tx}|C_j, Ht)}{P_i} \tag{7}$$

Here,  $P(\omega_{tx}|C_j, Ht)$  is estimated the probability of word  $\omega_{tx}$  (the  $x$ th word in slot  $t$ ) with class  $C_j$  and the type is  $Ht$ . For the avoidance of Zero probabilities, Laplace smoothing is used [75].

### 3.2 Data preprocessing

#### 3.2.1 Resampling with replacement method

The resampling with replacement method is a bootstrapping-based approach to create synthetic data. This method creates a number of random samples using replacement or without replacement methods. It solves class imbalance problem by influencing the original class distributions and producing more uniform class distributions. As,

in software fault prediction, defect prone and non-defect prone classes are not equal. Therefore, it creates more class instances by oversampling for a minority class. This method handles class imbalance problem by producing the uniform ratio for both classes which helps to reduce the bias of a classifier.

### 3.2.2 Smote

Synthetic Minority Oversampling Technique (SMOTE) is an oversampling technique used to handle class imbalance problem. SMOTE oversample the instances that belong to minority class to increase its instances. This method is based on K nearest neighbor approach. In this work, we have used  $k = 5$  for neighbor selection. Oversampling is done by taking a sample of a minority class and then creating synthetic sample along the direction of nearest neighbors. Chosen KNN depends upon the amount of oversampling required. For example, for 200% oversampling, only two nearest neighbors are selected from five nearest neighbors. The generation of the synthetic sample requires following two steps; calculate the distance between the selected sample that is under consideration and selected neighbors. Then multiply this difference by a random number between 0 and 1. Sum up the result and the selected sample under consideration. This causes selection of a random point between two selected samples. This approach forces decision boundary of the minority class to become more general. In this way, we have handled class imbalance problem in software fault detection datasets.

## 3.3 Feature selection method

### 3.3.1 Fisher linear discriminant analysis (FLDA)

Feature selection is a process for selecting most discriminative features. As we work on software fault prediction, datasets possess many features. The aim is to provide most discriminative features to machine learning classifiers in order to improve their performance. For this purpose, we have used dimensionality reduction technique. The dimensionality reduction technique serves two purposes; reduces dimensions of feature vector and select discriminative features. In this study, we have used FLDA for dimensionality reduction. It is a supervised dimensionality reduction approach that uses the class labels for identifying the most discriminative features [28]. Unlike unsupervised dimensionality reduction approaches like principal component analysis (PCA), it selects only those features that suit class labels. For software fault prediction, there are two classes; fault-prone and non-fault-prone. Therefore, FLDA only selects those features which are helpful to classify instances in abovementioned classes. FLDA converts high dimensional data into lower dimensional data by calculating scattered matrices within and between the class labels; represented by  $S_w$  and  $S_B$ , respectively. The transform matrix  $F_{FLDA}$  in the direction of  $W \in \mathbb{R}^n$  can be obtained as:

$$F_{FLDA} = \operatorname{argmax}_w \frac{|W^T S_B W|}{|W^T S_W W|} \quad (8)$$

**Table 2** Details for datasets used for software fault prediction

Dataset	Description	Project size	# faulty modules/total # modules	% faulty modules
Ar1		2467	9/121	7.4
Ar3		5624	8/64	12.5
AR4		9196	20/108	18.5
AR5		2732	8/36	22.2
AR6		2078	15/101	14.9
jEdit-4.0_4.2		56,502	134/274	48.9
jEdit-4.2_4.3		83,127	204/369	55.2
Kc1		42,965.1	326/2109	15.5
Kc2		19,259.1	107/522	20.4
Kc3		7749	43/458	9.4
Mc2		6134	52/161	32.3
Mw1		8341	31/403	7.7
Pc1		25,924.1	77/1109	6.9
Pc2		26,863	23/5589	0.4
Pc4		30,055	178/1458	12.2

The optimal maximizing solution can be calculated by solving this eigenvector problem:

$$S_B W = \Gamma S_W W \quad (9)$$

Here  $\Gamma$  represents the diagonal eigenvalue matrix.

## 4 Experimental setup

### 4.1 Datasets

In this study, we have gathered various datasets from PROMISE repository. This repository contains different publically available datasets and contains many datasets for software fault prediction from different open sources. We have used sixteen publically available datasets including Ar1, Ar3, AR4, AR5, AR6, CM1\_req, jEdit-4.0\_4.2, jEdit-4.2\_4.3, Kc1, Kc2, Kc3, Mc2, Mw1, Pc1, Pc2, and Pc4 (<http://promise.site.uottawa.ca/SERepository/datasets-page.html>). As one of the objectives of our study is to handle class imbalance data, therefore, we have intentionally used datasets with class imbalance data. Most of the datasets used in this study are badly affected by class imbalance problem. We have also tried to cover different sizes of datasets in terms of lines of code and number of instances to check the effectiveness of the proposed technique. Table 2 presents details regarding all datasets, and Table 3 presents software metrics used in these datasets.

**Table 3** Details for software metrics used for software fault prediction [34]

	Metrics	Representation
LOC based metrics	LOC total	LOC
	LOC blank	BLOC
	LOC executable	SLOC
	LOC comments	CLOC
	LOC code and comment	C&SLOC
	No of lines	nl
	Percent comment	% Comment
	Halsted metrics	Num operators
Num operands		$N2$
Num unique operators		$n1$
Num unique operands		$n2$
Length		$N = N1 + N2$
Difficulty		$D = \frac{n1+N2}{2+N2}$
Level		$1/D$
Programming effort		$E = D * V$
Programming time		$T = E/18$
Error estimate		$B = V/3000$
Content (intelligence) Vocabulary	$I = V/D$	
McCabe metrics	Cyclomatic complexity	$v(G)$
	Cyclomatic density	$vd(G)$
	Decision density	$dd(G)$
	Design complexity	$iv(G)$
	Design density	$id(G) = iv(G)/v(G)$
	Essential complexity	$ev(G)$
	Essential density	$ed(G) = \frac{ev(G)-1}{v(G)-1}$
	Global data density	$gd(G) = \frac{gdv(G)}{v(G)}$
	Norm cyclomatic complexity	$Normv(G) = \frac{v(G)}{nl}$
Maintenance severity	$Mainsev = \frac{ev(G)}{v(G)}$	
Miscellaneous	Branch count	Branch_C
	Call pairs	Call_C
	Condition count	Cond_C
	Decision count	Dec_C
	Edge count	Edge_C
	Node count	Node_C
	Parameter count	Parameter_C
	Modified condition count	Mod_Con_C

**Table 4** Confusion matrix: here each row represents actual class and each column represents predicted class

Actual	Predicted		
	Selected	Not selected	Total
Faulty modules	$N_{ms}$	$N_{mn}$	$N_m$
Non-faulty modules	$N_{cs}$	$N_{cn}$	$N_c$
	$N_s$	$N_n$	$N$

### 4.2 Evaluation metrics

To check the performance of given mispronunciation detection system, precision, recall, F1, AUC, and coverage are the factors being used as performance measures [29–32]. Firstly, divide phonemes into two classes for metrics computation; correctly classified and misclassified (refer to Table 4).

Precision is required so that we can determine a detected class by the machine is relevant or not. It also determines the effectiveness of fault detection system. Mathematically, it can be presented as:

$$Precision = \frac{N_{ms}}{N_s} \times 100\% \tag{10}$$

The recall is required to determine the probability that a relevant class is detected. Mathematically, it can be written as:

$$Recall = \frac{N_{ms}}{N_m} \times 100\% \tag{11}$$

In this equation,  $N_{ms}$ ,  $N_s$  and  $N_m$  show number of true faults spotted by the system, total number of faults detected by system and number of faults labeled by language experts, respectively. The *F1* measure is combinational. It combines precision and recall into a single metric and can be computed as:

$$Recall = \frac{N_{ms}}{N_m} \times 100\% \tag{12}$$

The system is also evaluated using AUC as it measures the potential for classification of correct and incorrect classes [33].

## 5 Results and discussion

This section represents results for all the experiments we have conducted in this study. First of all, we have reported results for the suitability of machine learning classifiers for software fault detection. Then, best four classifiers are selected by critically analyzing the performance of all classifiers for software fault prediction. These four classifiers are used further in all experiments with sampling techniques and FLDA.



Different experiments are conducted to validate the results of our proposed algorithm. These include the experiments using simple sampling techniques like SMOTE and Resample. Then, we have combined FLDA technique for feature selection and dimensionality reduction separately in combination with these sampling techniques. Results are then compared on the basis of precision, recall,  $f$ -measure and AUC to verify the effectiveness of the proposed method.

### 5.1 Results using different machine learning classifiers

In the first experiment, nine different machine learning classifiers are trained and tested for all datasets. These classifiers include Bayesian, Naïve Bayes, MLP, SVM, KNN, Adaboost, Bagging, Zero R and RF. These selections of these classifiers were based on the diverse underline assumptions. Bayesian and Naïve Bayes were selected because both these algorithms are based on very strong probability models. Generally, Naïve Bayes works well for classification problems. SVM is also used in this experiment due to its very good performance for binary classification and generalizability for different problems. We have also used ANN due to its good performance for classification problems. Ensemble methods, i.e., bagging and RF are also used to make comparisons fair among different types of classifiers. These performances of these classifiers are tested over eleven different software fault detection datasets.

The results of all these classifiers for all datasets are presented in Table 5, 6, 7 and 8 for precision, recall,  $f$ -measure and AUC, respectively. Generally, the performance of all these classifiers is good and results are comparable to each other except Zero R. Some of these classifiers NB, MLP, and RF show consistent performance for all datasets. The performance of all classifiers is not satisfactory for some datasets like jEdit\_4.0\_4.2 and jEdit\_4.2\_4.3 except for KNN. The KNN shows better results for these datasets because it predicts the target class by selecting nearest neighbors at runtime. The average classification results for Bayesian, NB, MLP, SVM, KNN, Adaboost, Bagging, Zero R and RF are 0.81, 0.83, 0.82, 0.82, 0.81, 0.80, 0.81, 0.69 and 0.83, respectively. As NB, MLP, and RF perform consistently well for all classifiers, therefore, we have used these three classifiers along with SVM for further experiments. SVM has been selected due to its suitability for binary classification problems.

### 5.2 Results using SMOTE and Resample

In this experiment, we have used two state-of-the-art sampling techniques to handle imbalance datasets. Many datasets used in this study are severely affected by class imbalance problem. SMOTE and Resample methods are most widely used to handle imbalance datasets for software fault prediction. We have used best four classifiers-NB, MPL, SVM, and RF to evaluate the results after applying SMOTE and Resample algorithms for all datasets. Results of these experiments are presented in Tables 9, 10, 11 and 12 for precision, recall,  $f$ -measure and AUC, respectively.

According to results, the performance of each classifier was significantly improved by using SMOTE and Resample. The average accuracies for NB, MLP, SVM and RF before using any method for imbalance data were 0.83, 0.82, 0.82 and 0.83, respec-

**Table 5** The results in the form of precision is shown here to check the suitability of different machine learning classifiers over various datasets

Classifiers	ar1	ar3	ar4	ar5	ar6	jEdit_4.0_4.2	jEdit_4.2_4.3	kc1	KC2	KC3	MC2	MW1	pc1	PC2	pc4
Bayes.Net	0.86	0.93	0.81	0.88	0.72	0.71	0.61	0.83	0.83	0.9	0.68	0.9	0.9	0.99	0.87
Naïve Bayes	0.90	0.92	0.83	0.85	0.81	0.72	0.65	0.86	0.82	0.88	0.73	0.9	0.89	0.99	0.86
MLP	0.89	0.93	0.80	0.71	0.81	0.71	0.62	0.84	0.834	0.86	0.68	0.9	0.92	0.99	0.89
SMO	0.86	0.87	0.84	0.83	0.89	0.70	0.64	0.82	0.85	0.85	0.72	0.85	0.87	0.99	0.89
IBK(KNN)	0.90	0.85	0.77	0.80	0.82	0.68	0.67	0.83	0.79	0.87	0.65	0.88	0.92	0.99	0.87
Adaboost MI	0.85	0.89	0.82	0.80	0.74	0.69	0.57	0.81	0.8	0.86	0.67	0.89	0.87	0.99	0.87
Bagging	0.86	0.84	0.79	0.74	0.88	0.72	0.67	0.84	0.82	0.86	0.71	0.88	0.93	0.99	0.89
Zero R	0.86	0.76	0.66	0.61	0.73	0.26	0.31	0.81	0.63	0.82	0.46	0.85	0.87	0.99	0.77
Random Forest	0.86	0.92	0.81	0.83	0.81	0.76	0.68	0.85	0.82	0.85	0.7	0.89	0.93	0.99	0.89

**Table 6** The results in the form of recall is shown here to check the suitability of different machine learning classifiers over various datasets

Classifiers	ar1	ar3	ar4	ar5	ar6	jEdit_4.0_4.2	jEdit_4.2_4.3	kc1	KC2	KC3	MC2	MW1	pc1	PC2	pc4
Bayes.Net	0.91	0.92	0.81	0.83	0.82	0.71	0.59	0.69	0.78	0.66	0.69	0.85	0.74	0.89	0.74
Naïve Bayes	0.85	0.91	0.84	0.83	0.82	0.69	0.53	0.82	0.84	0.85	0.74	0.83	0.89	0.97	0.87
MLP	0.90	0.94	0.81	0.69	0.82	0.71	0.62	0.86	0.85	0.87	0.69	0.92	0.94		0.89
SMO	0.92	0.89	0.85	0.83	0.87	0.68	0.64	0.85	0.83	0.9	0.72	0.92	0.93	0.99	0.89
IBK(KNN)	0.90	0.86	0.79	0.78	0.83	0.68	0.67	0.84	0.81	0.88	0.67	0.88	0.92	0.99	0.87
Adaboost M1	0.88	0.91	0.84	0.81	0.79	0.69	0.55	0.85	0.81	0.88	0.69	0.91	0.93	0.99	0.89
Bagging	0.91	0.87	0.82	0.75	0.89	0.72	0.68	0.86	0.84	0.84	0.72	0.92	0.94	0.99	0.9
Zero R	0.93	0.87	0.81	0.78	0.85	0.51	0.55		0.79	0.91	0.68	0.92	0.93	0.99	0.87
Random Forest	0.90	0.92	0.83	0.83	0.85	0.76	0.68	0.87	0.83	0.89	0.72	0.91	0.94	0.99	0.9
libsvm	0.91	0.92	0.81	0.83	0.82	0.71	0.59	0.85	0.79	0.91	0.67	0.91	0.94	0.99	0.88

**Table 7** The results in the form of  $f_1$ -measure is shown here to check the suitability of different machine learning classifiers over various datasets

Classifiers	ar1	ar3	ar4	ar5	ar6	jEdit_4.0_4.2	jEdit_4.2_4.3	kc1	KC2	KC3	MC2	MW1	pc1	PC2	pc4
Bayes.Net	0.88	0.92	0.81	0.84	0.79	0.71	0.59	0.73	0.79	0.74	0.68	0.87	0.803	0.94	0.78
Naïve Bayes	0.87	0.91	0.83	0.84	0.82	0.68	0.48	0.82	0.82	0.86	0.71	0.86	0.89	0.98	0.86
MLP	0.90	0.93	0.81	0.70	0.82	0.71	0.62	0.83	0.84	0.863	0.68	0.904	0.92		0.89
SMO	0.89	0.87	0.82	0.83	0.83	0.66	0.64	0.86	0.78	0.86	0.66	0.88	0.89	0.99	0.86
IBK(KNN)	0.90	0.85	0.78	0.79	0.82	0.68	0.67	0.84	0.8	0.87	0.66	0.88	0.92	0.99	0.87
Adaboost	0.87	0.89	0.83	0.80	0.77	0.69	0.55	0.81	0.81	0.87	0.66	0.9	0.89	0.99	0.87
M1															
Bagging	0.88	0.84	0.79	0.74	0.81	0.72	0.68	0.83	0.82	0.87	0.69	0.89	0.93	0.99	0.89
Zero R	0.89	0.81	0.73	0.68	0.78	0.35	0.39		0.7	0.86	0.55	0.89	0.89	0.99	0.82
Random Forest	0.88	0.92	0.81	0.83	0.81	0.76	0.68	0.85	0.84	0.87	0.69	0.89	0.93	0.99	0.89
LIBsvm	0.88	0.92	0.81	0.84	0.79	0.71	0.59	0.78	0.7	0.86	0.54	0.88	0.91	0.99	0.82

**Table 8** The results in the form of AUC is shown here to check the suitability of different machine learning classifiers over various datasets

Classifiers	ar1	ar3	ar4	ar5	ar6	jEdit_4.0_4.2	jEdit_4.2_4.3	kc1	KC2	KC3	MC2	MW1	pc1	PC2	pc4
Bayes.Net	0.39	0.74	0.77	0.38	0.38	0.78	0.63	0.79	0.82	0.81	0.66	0.75	0.70	0.79	0.83
Naïve Bayes	0.69	0.82	0.81	0.91	0.65	0.73	0.66	0.79	0.83	0.82	0.70	0.73	0.65	0.82	0.84
MLP	0.70	0.75	0.71	0.80	0.75	0.79	0.65	0.77	0.83	0.70	0.70	0.65	0.72	-	-
SMO	0.50	0.62	0.64	0.76	0.57	0.67	0.63	0.52	0.60	0.51	0.59	0.50	0.50	0.50	0.56
IBK(KNN)	0.77	0.67	0.59	0.75	0.67	0.68	0.67	0.74	0.64	0.73	0.58	0.61	0.74	0.85	0.72
Adaboost MI	0.66	0.70	0.86	0.81	0.59	0.80	0.57	0.78	0.78	0.76	0.59	0.75	0.80	0.82	0.91
Bagging	0.61	0.74	0.81	0.79	0.59	0.82	0.73	0.81	0.83	0.80	0.67	0.78	0.85	0.82	0.92
Zero R	0.44	0.37	0.49	0.37	0.40	0.49	0.49	0.50	0.49	0.47	0.48	0.48	0.49	0.45	0.50
Random Forest	0.77	0.79	0.81	0.86	0.67	0.83	0.75	0.82	0.83	0.85	0.70	0.77	0.88	0.81	0.95

**Table 9** The results in the form of precision is shown for SMOTE and Resample using machine learning classifiers over various datasets

Classifier	ar1	ar3	ar4	ar5	ar6	jEdit_4.0_4.2	jEdit_4.2_4.3	KC1	KC2	KC3	MC2	MW1	PC1	PC2	PC4
<b>SMOTE</b>															
NB	0.83	0.9	0.81	0.87	0.69	0.74	0.66	0.7	0.77	0.81	0.68	0.84	0.83	0.99	0.81
MLP	0.92	0.85	0.81	0.82	0.82	0.8	0.67	0.77	0.78	0.82	0.72	0.84	0.89	-	-
SMO	0.86	0.88	0.82	0.87	0.75	0.77	0.7	0.76	0.78	0.85	0.68	0.86	0.89	0.98	0.85
RF	0.87	0.9	0.86	0.84	0.84	0.84	0.76	0.86	0.84	0.89	0.81	0.87	0.92	0.99	0.93
<b>Resample</b>															
NB	0.91	0.89	0.88	0.9	0.84	0.69	0.62	0.81	0.8	0.89	0.71	0.9	0.91	0.99	0.87
MLP	0.95	0.96	0.88	0.9	0.9	0.76	0.62	0.86	0.84	0.95	0.82	0.93	0.95	-	-
SMO	0.95	0.84	0.85	0.87	0.73	0.69	0.59	0.83	0.82	0.9	0.76	0.85	0.93	0.99	0.91
RF	0.86	0.99	0.94	0.94	0.92	0.87	0.84	0.93	0.93	0.96	0.89	0.95	0.98	1	0.95

**Table 10** The results in the form of recall is shown for SMOTE and Resample using machine learning classifiers over various datasets

Classifier	ar1	ar3	ar4	ar5	ar6	jEdit_4.0_4.2	jEdit_4.2_4.3	KC1	KC2	KC3	MC2	MW1	PC1	PC2	PC4
<b>SMOTE</b>															
NB	0.77	0.9	0.8	0.86	0.73	0.68	0.66	0.77	0.77	0.81	0.63	0.86	0.85	0.97	0.82
MLP	0.85	0.86	0.82	0.82	0.81	0.8	0.68	0.79	0.79	0.83	0.72	0.86	0.9	-	-
SMO	0.85	0.89	0.82	0.86	0.77	0.74	0.66	0.77	0.79	0.84	0.68	0.87	0.87	0.99	0.85
RF	0.89	0.9	0.87	0.84	0.84	0.84	0.76	0.86	0.84	0.9	0.81	0.89	0.92	0.99	0.93
<b>Resample</b>															
NB	0.89	0.87	0.89	0.89	0.86	0.68	0.52	0.82	0.82	0.87	0.73	0.86	0.91	0.92	0.87
MLP	0.95	0.95	0.89	0.89	0.9	0.76	0.62	-	0.85	0.95	0.82	0.94	0.95	-	-
SMO	0.95	0.87	0.85	0.86	0.85	0.69	0.59	0.85	0.82	0.91	0.76	0.92	0.93	1	0.91
RF	0.97	0.98	0.94	0.94	0.92	0.87	0.84	0.94	0.93	0.96	0.89	0.96	0.98	1	0.95

**Table 11** The results in the form of *f-measure* is shown for SMOTE and Resample using machine learning classifiers over various datasets

Classifier	ar1	ar3	ar4	ar5	ar6	jEdit_4.0_4.2	jEdit_4.2_4.3	KC1	KC2	KC3	MC2	MW1	PC1	PC2	PC4
<b>SMOTE</b>															
NB	0.79	0.9	0.79	0.87	0.7	0.6	0.75	0.75	0.81	0.7	0.6	0.84	0.98	0.82	0.79
MLP	0.85	0.85	0.82	0.82	0.81	0.67	0.77	0.78	0.83	0.72	0.85	0.87	-	-	0.85
SMO	0.81	0.89	0.81	0.87	0.72	0.57	0.72	0.78	0.78	0.67	0.84	0.81	0.99	0.82	0.81
RF	0.87	0.9	0.86	0.84	0.82	0.76	0.86	0.84	0.89	0.81	0.87	0.91	0.99	0.93	0.87
<b>Resample</b>															
NB	0.9	0.88	0.88	0.89	0.85	0.46	0.82	0.8	0.88	0.7	0.88	0.91	0.96	0.87	0.9
MLP	0.95	0.95	0.89	0.89	0.89	0.62	-	0.82	0.95	0.82	0.93	0.94	-	-	0.95
SMO	0.94	0.84	0.82	0.87	0.78	0.56	0.79	0.76	0.88	0.76	0.89	0.91	0.99	0.88	0.94
RF	0.96	0.99	0.93	0.94	0.91	0.83	0.93	0.93	0.96	0.89	0.95	0.98	1	0.95	0.96



**Table 12** The results in the form of AUC is shown for SMOTE and Resample using machine learning classifiers over various datasets

Classifier	ar1	ar3	ar4	ar5	ar6	jEdit_4.0_4.2	jEdit_4.2_4.3	KC1	KC2	KC3	MC2	MW1	PC1	PC2	PC4
<b>SMOTE</b>															
NB	0.77	0.91	0.84	0.93	0.78	0.73	0.67	0.79	0.84	0.82	0.72	0.78	0.68	0.79	0.86
MLP	0.72	0.78	0.83	0.90	0.81	0.79	0.70	–	0.84	0.79	0.77	0.85	0.85	–	–
SMO	0.52	0.82	0.74	0.87	0.58	0.63	0.60	0.60	0.73	0.54	0.67	0.58	0.51	0.50	0.67
RF	0.90	0.87	0.89	0.95	0.85	0.89	0.82	0.91	0.90	0.92	0.88	0.89	0.94	0.95	0.97
<b>Resample</b>															
NB	0.57	0.83	0.77	0.95	0.82	0.65	0.70	0.78	0.87	0.85	0.74	0.78	0.67	0.86	0.88
MLP	0.84	0.74	0.76	0.94	0.92	0.80	0.71	–	0.86	–	0.79	0.83	–	–	–
SMO	0.50	0.73	0.48	0.71	0.58	0.54	0.57	0.51	0.70	0.58	0.60	0.50	0.50	0.50	0.61
RF	0.90	0.82	0.93	0.98	0.97	0.94	0.92	0.95	0.93	0.97	0.95	0.90	0.97	0.94	0.98

tively. The average accuracy has improved by using SMOTE, but decrease in precision was also observed in some cases for all four classifiers. Precision has improved for the datasets like jEdit\_4.0\_4.2 and jEdit\_4.2\_4.3 that were showing poor performance when no sampling techniques were applied. MLP and RF showed considerably best performance out of all four classifiers using SMOTE. Exactly similar cases were observed for recall and  $f$ -measure values.

Performance of each classifier has improved when Resample method is used with these classifiers. The results for all four classifiers are much better with Resample method as compared to SMOTE for all datasets. Results are more consistent and better of each classifier for fault prediction. Precision, recall,  $f$ -measure, and AUC have also improved by using these methods. Similar performance trends are observed for precision, recall,  $f$ -measure, and AUC. In some cases, NB performs better than remaining classifiers while it shows variations in results for some datasets. Generally, MLP and RF give a best overall performance for Resample method. It is because of a strong framework of MLP that can adjust input weights while calculating outputs. The performance for RF is also better because of its ensemble nature. Strangely, the performance for SVM classifier is overall not exceptional. The reason might be that SVM faces issue while handling imbalance data. The performance of these sampling methods is very good. However, all these classifiers are not able to outperform each other consistently. To handle these issues, further experiments were conducted that involve dimensionality reduction-based feature selection.

### 5.3 Results for SMOTE-FLDA and Resample-FLDA

It has been observed that results are not consistent with simple classification and sampling techniques as shown in Tables 13, 14, 15 and 16 for precision, recall,  $f$ -measure, and AUC, respectively. One of the reasons for this inconsistency is irrelevant and redundant features. The feature vector present in dataset consists of many features which do not play any part in classifier training. FLDA transforms high dimensional data into lower space by selecting the most discriminative features. This helps us to use only the important features to train classifiers. To make fair comparisons, we have used FLDA directly on the dataset without using any sampling technique. Then, we have applied FLDA after using both sampling techniques; SMOTE and Resample denoted as SMOTE-FLDA and Resample-FLDA, respectively. The results suggest Resample-FLDA gives exceptional results, while the performance for SMOTE-FLDA is not very good. The reason behind the low performance of SMOTE is that it creates and use synthetic data. Therefore, if the created data is wrong, the error will be propagated further.

The results for Resample-FLDA show consistent improvements in precision, recall,  $f$ -measure, and AUC for all datasets. Precision for each classifier against all datasets has been significantly improved. In previous experiments, results for jEdit\_4.0\_4.2, jEdit\_4.2\_4.3, kc1, and kc2 were not very good. By applying Resample-FLDA, results have improved especially for kc1 and kc2. The results, in comparison with simple SMOTE, Resample and FLDA were much better. The results for RF were better than remaining classifiers for kc1 and kc2 using SMOTE-FLDA and Resample-

**Table 13** The results in the form of precision is shown for FLDA, SMOTE-FLDA and Resample-FLDA using machine learning classifiers over various datasets

Classifiers	ar1	ar3	ar4	ar5	ar6	jEdit_4.0_4.2	jEdit_4.2_4.3	KC1	KC2	KC3	MC2	MW1	PC1	PC2	PC4
<b>FLDA</b>															
NB	0.93	0.91	0.85	0.94	0.92	0.74	0.65	0.85	0.83	0.87	0.73	0.89	0.91	0.83	0.93
MLP	0.93	0.97	0.89	0.94	0.89	0.73	0.62	0.84	0.83	0.81	0.76	0.9	0.92	0.96	0.93
SMO	0.86	0.9	0.89	0.97	0.92	0.73	0.59	0.87	0.85	0.78	0.7	0.9	0.87	0.9	0.86
RF	0.89	0.96	0.88	0.94	0.89	0.65	0.61	0.81	0.79	0.83	0.78	0.88	0.91	0.97	0.89
<b>SMOTE-FLDA</b>															
NB	0.95	0.94	0.9	0.96	0.94	0.74	0.71	0.78	0.81	0.89	0.79	0.91	0.86	0.88	0.95
MLP	0.92	0.96	0.87	0.96	0.93	0.73	0.69	0.77	0.81	0.88	0.78	0.91	0.87	0.99	0.92
SMO	0.93	0.96	0.89	0.98	0.92	0.73	0.72	0.78	0.81	0.88	0.79	0.9	0.76	0.99	0.93
RF	0.88	0.92	0.81	0.96	0.88	0.65	0.53	0.74	0.76	0.83	0.71	0.91	0.84	0.99	0.88
<b>Resample-FLDA</b>															
NB	0.96	0.98	0.88	0.1	0.93	0.72	0.64	0.86	0.84	0.91	0.83	0.92	0.86	0.87	0.96
MLP	0.97	0.98	0.91	0.1	0.94	0.76	0.65	0.85	0.85	0.94	0.89	0.94	0.87	0.99	0.97
SMO	0.95	0.95	0.91	0.1	0.94	0.73	0.68	0.87	0.85	0.89	0.81	0.95	0.76	0.99	0.95
RF	0.97	0.1	0.91	0.1	0.95	0.83	0.84	0.93	0.91	0.97	0.93	0.95	0.84	0.99	0.97

**Table 14** The results in the form of recall is shown for FLDA, SMOTE-FLDA and Resample-FLDA using machine learning classifiers over various datasets

Classifiers	ar1	ar3	ar4	ar5	ar6	jEdit_4.0_4.2	jEdit_4.2_4.3	KC1	KC2	KC3	MC2	MW1	PC1	PC2	PC4
<b>FLDA</b>															
NB	0.93	0.91	0.85	0.94	0.92	0.74	0.65	0.85	0.83	0.87	0.73	0.89	0.91	0.91	0.83
MLP	0.93	0.97	0.89	0.94	0.89	0.73	0.62	0.84	0.83	0.81	0.76	0.9	0.92	0.92	0.96
SMO	0.86	0.9	0.89	0.97	0.92	0.73	0.59	0.87	0.85	0.78	0.7	0.9	0.87	0.87	0.9
RF	0.89	0.96	0.88	0.94	0.89	0.65	0.61	0.81	0.79	0.83	0.78	0.88	0.91	0.91	0.97
<b>SMOTE-FLDA</b>															
NB	0.95	0.94	0.9	0.96	0.94	0.74	0.71	0.78	0.81	0.89	0.79	0.91	0.86	0.88	0.86
MLP	0.92	0.96	0.87	0.96	0.93	0.73	0.69	0.77	0.81	0.88	0.78	0.91	0.87	0.99	0.86
SMO	0.93	0.96	0.89	0.98	0.92	0.73	0.72	0.78	0.81	0.88	0.79	0.9	0.76	0.99	0.86
RF	0.88	0.92	0.81	0.96	0.88	0.65	0.53	0.74	0.76	0.83	0.71	0.91	0.84	0.99	0.82
<b>Resample-FLDA</b>															
NB	0.96	0.98	0.88	0.1	0.93	0.72	0.64	0.86	0.84	0.91	0.83	0.92	0.92	0.86	0.87
MLP	0.97	0.98	0.91	0.1	0.94	0.76	0.65	0.85	0.85	0.94	0.89	0.94	0.94	0.87	0.99
SMO	0.95	0.95	0.91	0.1	0.94	0.73	0.68	0.87	0.85	0.89	0.81	0.95	0.95	0.76	0.99
RF	0.97	0.1	0.91	0.1	0.95	0.83	0.84	0.93	0.91	0.97	0.93	0.95	0.95	0.84	0.99

**Table 15** The results in the form of  $f$ -measure is shown for FLDA, SMOTE-FLDA and Resample-FLDA using machine learning classifiers over various datasets

Classifiers	ar1	ar3	ar4	ar5	ar6	jEdit_4.0_4.2	jEdit_4.2_4.3	KC1	KC2	KC3	MC2	MW1	PC1	PC2	PC4
<b>FLDA</b>															
NB	0.93	0.91	0.85	0.94	0.92	0.74	0.65	0.85	0.83	0.87	0.73	0.89	0.91	0.91	0.83
MLP	0.93	0.97	0.89	0.94	0.89	0.73	0.62	0.84	0.83	0.81	0.76	0.9	0.92	0.92	0.96
SMO	0.86	0.9	0.89	0.97	0.92	0.73	0.59	0.87	0.85	0.78	0.7	0.9	0.87	0.87	0.9
RF	0.89	0.96	0.88	0.94	0.89	0.65	0.61	0.81	0.79	0.83	0.78	0.88	0.91	0.91	0.97
<b>SMOTE-FLDA</b>															
NB	0.95	0.94	0.9	0.96	0.94	0.74	0.71	0.78	0.81	0.89	0.79	0.91	0.86	0.88	0.86
MLP	0.92	0.96	0.87	0.96	0.93	0.73	0.69	0.77	0.81	0.88	0.78	0.91	0.87	0.99	0.86
SMO	0.93	0.96	0.89	0.98	0.92	0.73	0.72	0.78	0.81	0.88	0.79	0.9	0.76	0.99	0.86
RF	0.88	0.92	0.81	0.96	0.88	0.65	0.53	0.74	0.76	0.83	0.71	0.91	0.84	0.99	0.82
<b>Resample-FLDA</b>															
NB	0.96	0.98	0.88	0.1	0.93	0.72	0.64	0.86	0.84	0.91	0.83	0.92	0.92	0.86	0.87
MLP	0.97	0.98	0.91	0.1	0.94	0.76	0.65	0.85	0.85	0.94	0.89	0.94	0.94	0.87	0.99
SMO	0.95	0.95	0.91	0.1	0.94	0.73	0.68	0.87	0.85	0.89	0.81	0.95	0.95	0.76	0.99
RF	0.97	0.1	0.91	0.1	0.95	0.83	0.84	0.93	0.91	0.97	0.93	0.95	0.95	0.84	0.99

**Table 16** The results in the form of AUC is shown for FLDA, SMOTE-FLDA and Resample-FLDA using machine learning classifiers over various datasets

Classifiers	ar1	ar3	ar4	ar5	ar6	jEdit_4.0_4.2	jEdit_4.2_4.3	KC1	KC2	KC3	MC2	MW1	PC1	PC2	PC4
<b>FLDA</b>															
NB	0.93	0.75	0.85	0.94	0.92	0.74	0.65	0.85	0.83	0.87	0.73	0.89	0.91	0.91	0.83
MLP	0.93	0.74	0.89	0.94	0.89	0.73	0.62	0.84	0.83	0.81	0.76	0.9	0.92	0.92	0.96
SMO	0.86	0.53	0.89	0.97	0.92	0.73	0.59	0.87	0.85	0.78	0.7	0.9	0.87	0.87	0.9
RF	0.89	0.81	0.88	0.94	0.89	0.65	0.61	0.81	0.79	0.83	0.78	0.88	0.91	0.91	0.97
<b>SMOTE-FLDA</b>															
NB	0.96	0.75	0.87	0.99	0.88	0.81	0.67	0.81	0.85	0.89	0.79	0.91	0.68	0.88	0.86
MLP	0.92	0.78	0.91	1.00	0.85	0.79	0.67	0.80	0.85	0.88	0.78	0.91	0.80	0.99	0.86
SMO	0.50	0.56	0.65	0.94	0.70	0.73	0.56	0.50	0.68	0.88	0.79	0.9	0.50	0.99	0.86
RF	0.89	0.84	0.88	0.94	0.76	0.74	0.66	0.73	0.76	0.83	0.71	0.91	0.84	0.99	0.82
<b>Resample-FLDA</b>															
NB	0.96	0.75	0.87	0.99	0.88	0.81	0.67	0.81	0.85	0.91	0.83	0.92	0.68	0.86	0.87
MLP	0.92	0.78	0.91	1.00	0.85	0.79	0.67	0.80	0.85	0.94	0.89	0.94	0.80	0.87	0.99
SMO	0.50	0.56	0.65	0.94	0.70	0.73	0.56	0.50	0.68	0.89	0.81	0.95	0.50	0.76	0.99
RF	0.89	0.84	0.88	0.94	0.76	0.74	0.66	0.73	0.76	0.97	0.93	0.95	0.84	0.84	0.99

FLDA. The precision for remaining three classifiers was under 0.80. The results for Resample-FLDA method produced minimum precision of 0.83 using NB and MLP. The remaining classifiers performed even better than these values.

The two proposed methods, SMOTE-FLDA and Resample-FLDA, showed great results. Results for precision, recall, and  $f$ -measure have been improved remarkably when the comparison is made with previous methods. SMOTE-FLDA performed exceptionally well for all datasets that show the performance of the system is improved by applying FLDA after SMOTE. All four classifiers showed consistent performance. The results for Resample-FLDA are even better than SMOTE-FLDA. As we discussed earlier that performance of Resample method as a sampling technique is better than SMOTE, same performance is continued here when we applied FLDA after aforementioned sampling techniques separately. As simple Resample method performs better than SMOTE, Resample-FLDA performs better than SMOTE-FLDA. The overall good performance of our proposed methods proves that there is a strong need to handle imbalance data, but it should be followed by dimensionality reduction and feature selection approaches. This gives us better and consistent performance for many classifiers over dataset with different sizes.

The improvement in performance of machine learning classification algorithms is because of FLDA. As the datasets are severely affected by class imbalance problem, classifiers are unable to correctly classify minor class instances. FLDA helps to solve this issue because it suits problem areas where a number of instances are less than the number of features. Therefore, use of FLDA after applying SMTOE and Resample improve the performance of classifiers. However, the performance of Resample method performs better as compared to SMOTE when it is used separately or in combination with FLDA.

A comprehensive comparison is also presented in Table 17 with some state-of-the-art techniques. The results are compared mostly using average values of AUC. The reported results of this comparison are also average precision, recall,  $f$ -measure and AUC. The results for proposed system researchers have used a small number of datasets, but instance reduction is not handled properly in most of the studies. Some researchers have reported better results than proposed systems, but size and number of datasets used are not large. In our proposed system, we have handled instance reduction and class imbalance problem efficiently and reported a good performance as compared to individual classifiers. We have also yielded better or comparable results for even large datasets. The performance of the proposed system for such large number of diverse datasets is very good.

## 6 Conclusion

SQA is a vital yet an expensive part of software life cycle. Identification of software faults before the testing phase can save a lot of maintenance cost and time. The software fault prediction helps to identify fault-prone modules using software metrics. The main aim of this study was to explore the performance of machine learning classifiers for software fault prediction over small-, medium- and large-size software. Nine most widely used machine learning classifiers are used for this purpose. Mostly classifiers

**Table 17** A comparative analysis of the proposed work with state-of-the-art system

Sr. #	Technique	Dataset	Techniques	Performance evaluation					
				Precision	Recall	F-measure	AUC	Sensitivity	Specificity
1	Ruchika Malhotra [35]	AR1	LR, ANN, SVM	N/A	N/A	N/A	N/A	73.39	49.68
2	Ruchika Malhotra [36]	KC1, Ivy, JEdit	NB, BN, RF, SVM, MLP	N/A	N/A	N/A	0.718	0.724	0.676
3	Ruchika Malhotra [37]	SF13, SF20, SF21, SF37, SF38, SF40, SF43	RF, MLP, NB, BN, SVM	91.68	N/A	N/A	0.948	N/A	N/A
4	Ruchika Malhotra [38]	CMI, Cos, Eclipse 2.0a	RF, SVM, RF	96%	N/A	N/A	N/A	N/A	N/A
5	Wasif Afzal [39]	jEdit, AR5, MC1, CMI, KC1_Mod	NB, PCA, IG, RLF, CFS	N/A	0 in 6	1 in 6	0.816	N/A	N/A
6	Raed Shatmawi [40]	Ant1.4, Ant1.5	LR, NB	N/A	0 in 6	1 in 6	0.695	N/A	N/A
7	Tao Wang [22]	CMI, JMI, KC1, KC2, KC3, KC4, MC1, MC2, MW1, PC1, PC2, PC3, PC4, PC5	Bagging, Adaboost, M1, NB, RF	N/A	N/A	N/A	0.75	N/A	N/A
8	Proposed method	Ar1, Ar3, AR4, AR5, AR6, CMI_req, jEdit-4.0_4.2, jEdit-4.2_4.3, Kc1, Kc2, Kc3, Mc2, Mw1, Pc1, Pc2, and Pc4	NB, MLP, SVM, RF	0.84	0.84	0.84	0.86	N/A	N/A



give poor performance for large datasets. The poor performance of these classifiers is due to class imbalance problem and irrelevant software metrics that are used to train classifiers. To overcome both of these issues, FLDA is used in combination with two sampling techniques, i.e., SMOTE and Resample with replacement. The sampling techniques handle the class imbalance problem and FLDA reduces dimensions while selecting best suitable software metrics for fault prediction. After applying these techniques, top four classifiers are used to evaluate the effectiveness of proposed methods over 15 publically available datasets. Precision, recall,  $f$ -measure, and AUC are used as performance measure.

In this study, many experiments were conducted to evaluate the effectiveness of proposed methods. The performance of machine learning classifiers got improved by using sampling techniques alone. Resample with replacement performed better than SMOTE. The best results were achieved when FLDA has been used after both of the sampling techniques separately. Resample-FLDA produced best results and outperformed all other methods (simple Resample, simple SMOTE, and SMOTE-FLDA). The reported results are better than many existing methods for software fault prediction. In future, we are planning to design an efficient method to predict a number of faults that can help to identify the resources required to handle these faults.

**Acknowledgements** This research was supported by Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education (NRF-2016R1D1A1A09919551).

## References

1. Andr B et al (2010) A symbolic fault-prediction model based on multiobjective particle swarm optimization. *J Syst Softw* 83(5):868–882
2. Manjula C, Florence L (2018) Deep neural network based hybrid approach for software defect prediction using software metrics. *Cluster Comput*. <https://doi.org/10.1007/s10586-018-1696-z>
3. Rathore SS, Kumar S (2017) Towards an ensemble based system for predicting the number of software faults. *Expert Syst Appl* 82(Supplement C):357–382
4. Aljamaan HI, Elish MO (2009) An empirical study of bagging and boosting ensembles for identifying faulty classes in object-oriented software. In: *IEEE Symposium on Computational Intelligence and Data Mining, 2009. CIDM'09. IEEE*
5. Chiu K-C, Huang Y-S, Lee T-Z (2008) A study of software reliability growth from the perspective of learning effects. *Reliab Eng Syst Saf* 93(10):1410–1421
6. Gao K et al (2011) Choosing software metrics for defect prediction: an investigation on feature selection techniques. *Softw Pract Exp* 41(5):579–606
7. Gray AR, MacDonell SG (1997) A comparison of techniques for developing predictive models of software metrics. *Inf Softw Technol* 39(6):425–437
8. Sharma D, Chandra P (2018) Software fault prediction using machine-learning techniques. In: *Smart computing and informatics*. Springer, pp 541–549
9. Muhamad FPB, Siahaan DO, Fatichah C (2018) Software fault prediction using filtering feature selection in cluster-based classification. *IPTEK Proc Ser* 4(1):59–64
10. Hall T et al (2012) A systematic literature review on fault prediction performance in software engineering. *IEEE Trans Softw Eng* 38(6):1276–1304
11. Helmer G et al (2007) Software fault tree and coloured Petri net—based specification, design and implementation of agent-based intrusion detection systems. *Int J Inf Comput Secur* 1(1–2):109–142
12. Graves TL et al (2000) Predicting fault incidence using software change history. *IEEE Trans Softw Eng* 26(7):653–661

13. Bell RM, Ostrand TJ, Weyuker EJ (2006) Looking for bugs in all the right places. In: Proceedings of the 2006 International Symposium on Software Testing and Analysis. ACM
14. Weyuker EJ, Ostrand TJ, Bell RM (2007) Using developer information as a factor for fault prediction. In: International Workshop on Predictor Models in Software Engineering. PROMISE'07: ICSE Workshops 2007. IEEE
15. Ostrand TJ, Weyuker EJ, Bell RM (2004) Where the bugs are. *SIGSOFT Softw Eng Notes* 29(4):86–96
16. Janes A et al (2006) Identification of defect-prone classes in telecommunication software systems using design metrics. *Inf Sci* 176(24):3711–3734
17. Mısırlı AT, Bener AB, Turhan B (2011) An industrial case study of classifier ensembles for locating software defects. *Softw Qual J* 19(3):515–536
18. Zheng J (2010) Cost-sensitive boosting neural networks for software defect prediction. *Expert Syst Appl* 37(6):4537–4543
19. Twala B (2011) Software faults prediction using multiple classifiers. In: 2011 3rd International Conference on Computer Research and Development (ICCRD). IEEE
20. Xu W et al (2011) Oncometabolite 2-hydroxyglutarate is a competitive inhibitor of  $\alpha$ -ketoglutarate-dependent dioxygenases. *Cancer Cell* 19(1):17–30
21. Khoshgoftaar TM, Geleyn E, Nguyen L (2003) Empirical case studies of combining software quality classification models. In: Third International Conference on Quality Software, 2003. Proceedings. IEEE
22. Wang S, Yao X (2013) Using class imbalance learning for software defect prediction. *IEEE Trans Reliab* 62(2):434–443
23. Kamei Y et al (2007) The effects of over and under sampling on fault-prone module detection. In: First International Symposium on Empirical Software Engineering and Measurement, 2007. ESEM 2007. IEEE
24. He P et al (2015) An empirical study on software defect prediction with a simplified metric set. *Inf Softw Technol* 59:170–190
25. Galar M et al (2012) A review on ensembles for the class imbalance problem: bagging-, boosting-, and hybrid-based approaches. *IEEE Trans Syst Man Cybern Part C (Appl Rev)* 42(4):463–484
26. Rodriguez D et al (2014) Preliminary comparison of techniques for dealing with imbalance in software defect prediction. In: Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering. ACM
27. Moser R, Pedrycz W, Succi G (2008) A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction. In: Proceedings of the 30th International Conference on Software Engineering. ACM, Leipzig, pp 181–190
28. Alexandre-Cortizo E, Rosa-Zurera M, Lopez-Ferreras F (2005) Application of fisher linear discriminant analysis to speech/music classification. In: EUROCON 2005-The International Conference on Computer as a Tool. IEEE
29. Al Hindi A et al (2014) Automatic pronunciation error detection of nonnative Arabic Speech. In: 2014 IEEE/ACS 11th International Conference on Computer Systems and Applications (AICCSA). IEEE
30. Franco H et al (1999) Automatic detection of phone-level mispronunciation for language learning. In: EUROSPEECH
31. Strik H et al (2009) Comparing different approaches for automatic pronunciation error detection. *Speech Commun* 51(10):845–852
32. Truong K et al (2004) Automatic pronunciation error detection: an acoustic-phonetic approach. In: InSTIL/ICALL Symposium 2004
33. Ghazanfar MA (2015) Experimenting switching hybrid recommender systems. *Intell Data Anal* 19(4):845–877
34. Singh P et al (2017) Fuzzy rule-based approach for software fault prediction. *IEEE Trans Syst Man Cybern Syst* 47(5):826–837
35. Malhotra R (2014) Comparative analysis of statistical and machine learning methods for predicting faulty modules. *Appl Soft Comput* 21:286–297
36. Malhotra R, Pritam N, Singh Y (2014) On the applicability of evolutionary computation for software defect prediction. In: 2014 International Conference on Advances in Computing, Communications and Informatics (ICACCI). IEEE
37. Malhotra R (2015) A systematic review of machine learning techniques for software fault prediction. *Appl Soft Comput* 27:504–518

38. Malhotra R, Bansal AJ (2015) Fault prediction considering threshold effects of object-oriented metrics. *Expert Syst* 32(2):203–219
39. Stanic B, Afzal W (2017) Process metrics are not bad predictors of fault proneness. In: *The 2017 IEEE International Workshop on Software Engineering and Knowledge Management SEKM'17*, 25 July 2017, Prague, Sweden
40. Shatnawi R (2017) The application of ROC analysis in threshold identification, data imbalance and metrics selection for software fault prediction. *Innov Syst Softw Eng* 13(2–3):201–217