



# Improved Software Fault Prediction Model Based on Optimal Features Set and Threshold Values Using Metaheuristic Approach

Manpreet Singh<sup>1</sup> · Jitender Kumar Chhabra<sup>1</sup>

Received: 28 February 2023 / Accepted: 2 August 2023  
© The Author(s), under exclusive licence to Springer Nature Singapore Pte Ltd 2023

## Abstract

Software fault prediction models are very important to prioritize software classes for effective testing and efficient use of resources so that the testing process's time, effort, and cost can be reduced. Fault prediction models can be based on either metrics' threshold values or machine learning. Code metrics' threshold-based models are easy to automate and faster than machine learning-based models, which can save significant time in the testing process. ROC, Alves ranking, and VARL are famous threshold value calculation techniques. Out of which ROC is the best threshold calculation technique. This research article proposes a new threshold values calculation technique based on metaheuristics. A genetic algorithm and particle swarm optimizer are used to calculate the threshold values, and the proposed technique is tested on ten open-source object-oriented software datasets and four open-source procedural software datasets. Results show that the metaheuristic-based thresholds give better results than ROC-based thresholds.

**Keywords** Software fault prediction · Metrics' threshold · PSO · GA · ROC

## Introduction

Software systems must be fault free to prevent the interruption of the service and money losses. Therefore, proper software testing after development is necessary to develop quality software, but testing of the software consumes almost half of the allocated resources and money [1]. Testing effort and cost should be reduced to develop quality software with a limited budget. High-quality software must have less number of faults, which reduces the testing cost and effort for detecting and correcting faults because the fault proneness of the modules depends upon the code quality of the software [1]. To develop high-quality software, all the software modules under development must be tested properly. However, it is difficult to execute all test cases based on execution paths of complex software to ensure it is defects free.

Fault prediction models based on code quality can identify poor-quality and fault-prone modules to support the testing and development team so that high-quality software can be developed with limited resources and budget. All the fault prediction models use code quality metrics to identify fault-prone modules of the software.

Researchers in the literature have proposed many code metrics-based defect prediction models. The Chidamber and Kemerer (CK) metrics suit is widely used for software defect prediction models in the object-oriented system [2–5]. These fault prediction models are either based on machine learning (ML) algorithms or on the code metrics' threshold values [6–13]. Code metrics' threshold-based fault prediction is faster than machine learning-based algorithms because the tester can directly identify the fault-prone modules by checking the threshold values. Code metrics' thresholds-based models predict software modules as defective when the value of a certain metric exceeds the threshold value. Many researchers proposed different defect prediction models based on threshold values of code metrics. Henderson-Sellers proposed a model to categorize the classes into safe, flag, and alarm (defective) based on different threshold values of code metrics [14]. He also suggested that thresholds of metrics are relative to the complexity of the software. Daly et al. [15] studied the average time needed for the

---

This article is part of the topical collection “Research Trends in Computational Intelligence” guest edited by Anshul Verma, Pradeepika Verma, Vivek Kumar Singh and S. Karthikeyan.

---

✉ Manpreet Singh  
manibhangu92@gmail.com

<sup>1</sup> Computer Engineering Department, National Institute of Technology, Kurukshetra, India

maintenance change in a software module with a different number of inheritance levels. He noted that up to three levels of inheritance reduce the time of maintenance change in software, and more than three levels of inheritance increase the time of maintenance change. Cartwright et al. [16] replicated the study done by Daly with different datasets, and the results differed from Daly's study. El Emam et al. [17, 18] proposed a new method to find the threshold values of code metrics which uses logistic regression to draw a threshold line. Erni and Lewerentz [19] proposed a method to find the thresholds based on the mean and standard deviation of code metrics' values. Shatnawi et al. [3] proposed a new method based on ROC curves to find the thresholds of different code metrics. Many other methods exist to find code metrics thresholds, like Alves ranking and VARL [20, 21]. All these methods calculate the threshold value for each code metric independently, but the software module cannot be categorized as fault based on only one metric's threshold value, and there is a need for relative threshold values of all metrics in the dataset.

This research article proposes a new method based on metaheuristic algorithms to find the thresholds of different code metrics in an object-oriented and procedural system. A new objective function is generated based on specificity (also called true negative rate) and sensitivity (also called true positive rate), and metaheuristic algorithms such as PSO and GA are used to optimize the objective function. The class with at least one metric value beyond the threshold value is considered faulty while calculating the TPR and TNR. We use the genetic algorithm and particle swarm algorithm to find the thresholds of code metrics and compare the performance of metaheuristic-based models with the ROC curves-based model, as Boucher et al. [1] proved that ROC curves-based models give the best performance as compared to other models such as Alves ranking and VARL.

The research article is organized into related works, methodology, results, and conclusion. Related works give a summary of past work done in this field. "Methodology" presents our proposed metaheuristic-based and ROC-based threshold calculation techniques. It also explains the feature selection procedure to select the optimal set of features to generate a threshold-based fault prediction model. The result section compares the performance of the metaheuristic-based model with the ROC curves-based model, followed by a conclusion.

## Related Works

Many studies have been done in the recent past to develop threshold-based software fault prediction models because there is a direct relationship between code quality and metrics

values. This research article section presents past studies to develop threshold-based software fault prediction models.

McCabe [22] proposed a threshold value for McCabe's cyclomatic complexity metric based on some experience. Catel et al. [11] used PREDICTIVE to calculate the thresholds of code quality metrics. The PREDICTIVE tool is no longer available. Rosenberg et al. [23] proposed a statistical analysis method to calculate the thresholds. They calculated the thresholds of code quality metrics based on experience and statistical methods for CK metrics suit.

Shatnawi et al. [3] proposed a new method that uses the ROC curve to find the thresholds of different code quality metrics. They plot a curve between sensitivity and specificity for different thresholds for software code metrics, and the threshold with the maximum area under the curve is chosen for software fault prediction. The experiment is performed on three versions of Eclipse projects for binary and multi-class classification. They proved that the proposed method is good for multi-class classification. However, for binary class classification, there is a need for more accurate and sophisticated methods.

Bender [20] proposed a novel method called VARL to calculate the thresholds of code metrics. This method was originally not proposed for software code metrics. However, it is used by many researchers in their studies to find the optimal threshold values [3, 6, 24]. This method uses univariate logistic regression to calculate the correct threshold of the metric. Some studies proved it useful for threshold calculation [24]; on the other hand, some researchers mentioned that no valid threshold values were found with VARL [3, 6].

Alves et al. [21] proposed a sophisticated method called Alves ranking to calculate the correct threshold of different code metrics. They combine the data of several projects to find the single threshold value per code metrics which is universal for all projects. They calculated the threshold value to find the quality of the software module. However, they have not used it to identify the defective modules of the software.

In their study, Benlarbi et al. [25] used logistic regression to find the threshold of software code metrics. They proved that logistic regression is not able to calculate the optimal thresholds and hence does not improve the model's prediction performance.

In their other study, Catel et al. [26] used the ROC curve method, originally proposed by Shatnawi et al. [3], to find the thresholds for software fault prediction. They modified this method a little bit. Instead of maximizing the value of sensitivity (TPR) and specificity (TNR), they plotted a curve between sensitivity and 1-specificity, and the AUC value among three points (0, 0), (sensitivity, 1-specificity), and (1, 1) is maximized.

After studying different techniques to find the threshold values in literature, we propose a metaheuristic-based method to calculate the relative thresholds of different code

metrics in this research article. Table 1 presents a comparative study of the proposed work with existing threshold-based fault prediction methods.

Based on the literature, we identified that threshold-based software fault prediction models are faster than machine learning-based models [1]. Once the universal threshold value of the code metric is decided, then it can be applied to any software to identify the faulty modules, which saves the cost and effort of the testing team because the tester can identify faulty modules immediately based on the code metric threshold values and no need of prediction model. Many techniques exist to calculate the thresholds of the code metrics, such as ROC, VARL, and Alves ranking. All these techniques calculate threshold values of the code metrics independently. However, there is a need to find the relative threshold values of code metrics because independent thresholds of the code metrics cannot identify the faulty classes.

### Methodology

This section of the research article shows the detailed steps of our proposed technique to decide the threshold values of different code metrics for software fault identification. Figure 1 shows the overall structure of the proposed model.

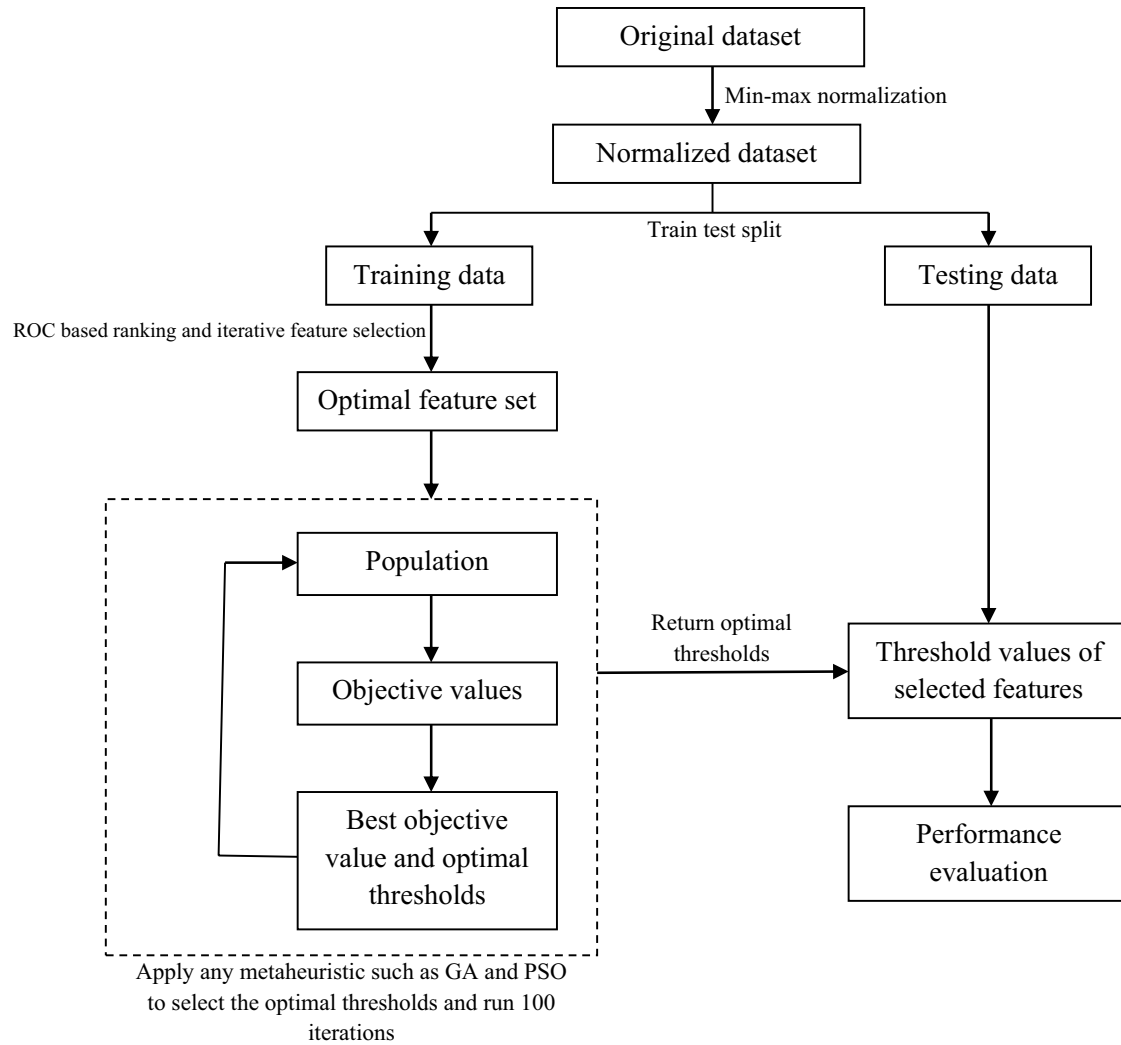
The first step of Fig. 1 normalize the original dataset using the min–max normalization technique. Normalized dataset are divided into training and testing data. In the training data, ranking is assigned to all the features based on the ROC value of the features. Feature with high ROC value has a higher rank. Relevant features are selected based on the iterative feature elimination method from the ranked features. In the iterative feature elimination process feature with the lowest rank is eliminated in each step. In the next step, a dataset with selected features is provided as input to train the model based on metaheuristic algorithms such as PSO and GA; after training the model based on metaheuristic algorithms, optimal threshold values of the selected features are extracted and used for software fault prediction on testing data. Finally, the model's performance is evaluated based on accuracy, sensitivity (true positive rate), and specificity (true negative rate).

### ROC Curve Based Thresholds

The ROC curve-based threshold detection method plots ROC curves for all code metrics. Optimal threshold values are selected where the sum of the sensitivity and specificity is maximum. Plotting the ROC curve needs one continuous variable and one categorical variable. Here continuous variable is the code metric, and the categorical variable is the class labels. The ROC curve is drawn based on a range of threshold values from the minimum possible value to the maximum possible value of the code metric.

**Table 1** A comparative study of proposed model with existing threshold based fault prediction models

Author	Methodology	Strength	Weakness
Shatnawi et al. [3]	ROC-based threshold calculation of code metrics	A new method based on the ROC curve is proposed to calculate the metrics threshold values and prove that threshold-based software fault prediction is faster and saves more time and cost	It is weaker than machine learning, and features should be very good to differentiate faulty and non-faulty modules based on binary split
Boucher et al. [1]	A comparative study of ROC, Alves ranking, and VARL techniques to calculate thresholds of the code metrics	It is proved that ROC is the best technique for threshold calculation, and in some cases, threshold-based models outperform machine learning algorithms	In most cases, machine learning provides better results than threshold-based fault prediction
Arar et al. [33]	Bug severity prediction based on threshold values calculated based on ROC and VARL	Threshold values can be used for multi-class classification to predict the severity of the fault	Machine learning is better for bug severity prediction
Proposed work	Metaheuristic-based threshold detection model for fault prediction	It performs better than the ROC method with less number of features. So, it saves more testing time and cost	Still, results are not better than advanced machine learning techniques such as bagging and boosting



**Fig. 1** Proposed fault prediction model based on metrics' thresholds

After drawing the ROC curve threshold value is chosen where the sum of sensitivity and specificity is maximum. These threshold values are used for software fault prediction after finding the threshold values of all code metrics. Class is considered faulty if the threshold value of at least one code metric is violated.

### Metaheuristic Algorithms Based Thresholds

Steps to apply metaheuristic algorithms to find the optimal threshold values of software code metrics are shown in algorithm 1.

---

#### Algorithm 1: Pseudo code of Proposed technique

---

**Input:**

**X:** Training Samples  $X \in \mathbb{R}^{n \times m}$

**y:** Class labels  $y \in \{0, 1\}^n$

**Step 1:** Dataset is normalized between 0 and 1 to standardize the ranges of code metrics.

**Step 2:** Best features are selection based on iterative feature elimination technique.

**Step 3:** Optimization algorithms such as GA and PSO are applied to calculate thresholds of code metrics.

**Step 4:** Optimal thresholds of the selected code metrics are returned after completion of all steps.

---

### Normalization

Simple min–max normalization is used to standardize the range of all selected code metrics to apply metaheuristic algorithms. Values of code metrics are normalized between 0 and 1.

Some code metrics, such as WMC, CBO, and RFC, are directly proportional to quantifying code quality. On the other hand, some code metrics, such as TCC, LCC, and SCOM, are inversely proportional to quantifying the code quality. It is required to reverse the values of inversely proportional code metrics by subtracting each value from 1.0 after normalization.

### Feature Selection

In the first step, we select relevant code metrics out of available 20 code metrics in promise repository datasets. The selection of relevant code metrics is very important to build a software fault prediction model based on metrics threshold values. Otherwise, irrelevant features can drastically reduce the software fault prediction performance. Features can be selected based on experimentation or feature selection techniques.

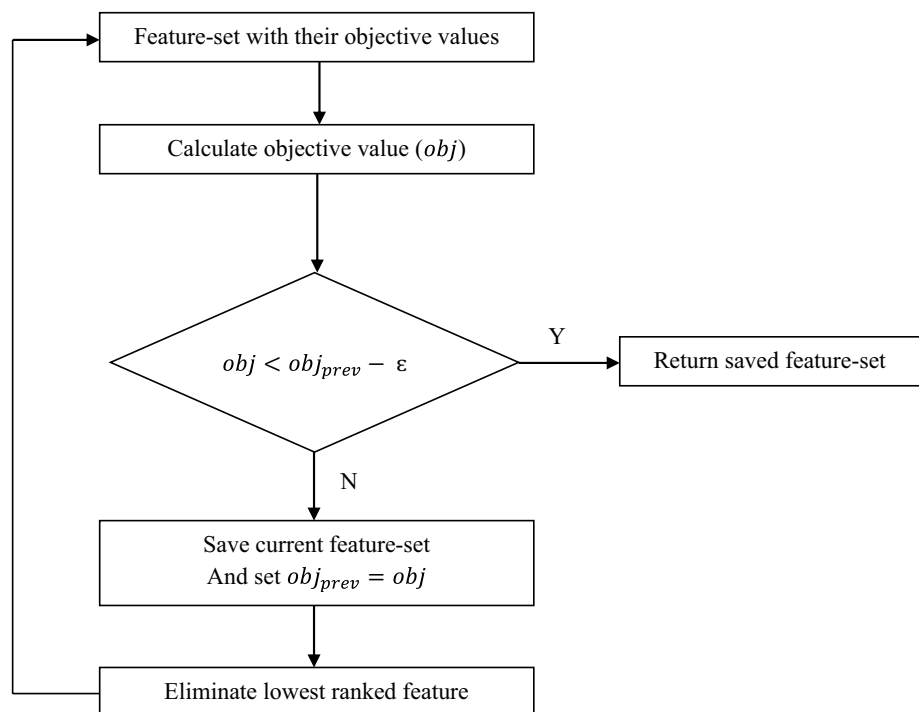
In this work, we select the best features based on the following steps:

- (1) The objective value (sum of specificity and sensitivity) of each performance metric on each selected dataset is calculated based on the ROC curve method.

- (2) The average object value of each performance metric on all selected datasets is taken to find the overall performance of each performance metric on all selected datasets.
- (3) After finding the average of objective values of all performance metrics on selected datasets, metrics are arranged in decreasing order based on average objective values because our objective function is a maximization function.
- (4) After sorting features in decreasing order, features are eliminated from lower to higher ranks based on their performance. In each iteration, the lowest-ranked feature (with the lowest objective value) is eliminated, and after eliminating the lowest-ranked feature, the combined objective value of the remaining features is calculated and compared with the performance of the previous objective value. If performance is degraded, then we stop further feature elimination.

Figure 2 shows the forth stem of the above procedure for the selection of features. In Fig. 2,  $obj$  represents the objective value of the feature set in the current iteration and  $obj_{prev}$  represents the objective value of the feature set in the previous iteration. Initially,  $obj_{prev}$  is set to 0 and feature set contains all 20 features arranged in descending order of ROC value.  $\epsilon$  is a small constant value that is set to 0.01 in our case.

**Fig. 2** Feature set selection based on iterative feature elimination



## Objective Function

Our objective is to maximize the sum of sensitivity and specificity. Sensitivity also called true positive rate (TPR), is used to check the number of correctly classified instances as positive out of total positive instances. Specificity also called true negative rate (TNR), is used to check the number of correctly classified instances as negative out of total negative instances. Sensitivity and specificity can be calculated based on Eqs. (1) and (2) [28].

$$\text{Sensitivity (TPR)} = \frac{\text{TP}}{\text{TP} + \text{FN}} \quad (1)$$

$$\text{Specificity (TNR)} = \frac{\text{TN}}{\text{TN} + \text{FP}} \quad (2)$$

Threshold values are used to differentiate the positive and negative classes, so we need to maximize the sum of sensitivity and specificity, as shown in Eq. (3).

$$\text{Objective function} = \left( \frac{\text{TP}}{\text{TP} + \text{FN}} + \frac{\text{TN}}{\text{TN} + \text{FP}} \right) \quad (3)$$

contain a value between 0 and 1 because each code metric is normalized between 0 and 1. A population of 20 chromosomes is generated in the first step of the algorithm. After generating the population, in the second step, the value of the objective function is calculated based on Eq. (3). To calculate the objective value, first, we need to calculate the value of specificity and sensitivity. Class with at least one metric value greater than the threshold value is considered faulty in calculating the sensitivity and specificity. After finding each chromosome's objective value, the chromosomes are sorted based on the high to low objective value. The best two chromosomes are selected, and a binary crossover is applied to generate the offspring. The best chromosome and its objective value are stored as the global optimum value. The mutation is performed to add randomization with a mutation probability of 0.1. After a hundred iterations of the algorithm, the best chromosome is returned, representing the threshold values of code metrics that can be used for software fault prediction. Algorithm 2 represents the overall steps of the genetic algorithm.

---

### Algorithm 2: Pseudo code of GA based threshold calculation

---

#### Input:

**X:** Training Samples  $\mathbf{X} \in \mathbb{R}^{n \times m}$

**y:** Class labels  $\mathbf{y} \in \{0, 1\}^n$

**Step 1:** Initialize random population.

**Step 2:** calculate the objective value of each chromosome.

**Step 3:** find two best chromosomes with maximum value of objective function.

**Step 4:** Apply binary crossover on best selected chromosomes.

**Step 5:** Identify the best objective value out of all four candidates. (two parents and two offspring)

**Step 6:** Update the global objective value and save the chromosome with best objective value.

Step 7: repeat steps 2 to 6 hundred times.

**Step 8:** return best chromosome after 50 iterations.

---

Class with metric value greater than threshold for at least one metric is considered as faulty otherwise class is considered as healthy. Value of objective function can vary between 0 and 2.

## Metaheuristic Algorithms

Any metaheuristic algorithm can optimize the objective function in Eq. (3). This research article uses a genetic algorithm (GA) and PSO (particle swarm optimizer) to optimize this objective function.

In the genetic algorithm, chromosome size is equals to the number of code metrics to detect threshold values. Each chromosome element represents the threshold value of the corresponding code metric, and each element must

Twenty random particle positions are initialized between 0 and 1 in the particle swarm optimizer. Each particle's size equals the number of code metrics in the feature set. All particles' velocities are initialized between  $-0.5$  and  $0.5$  for each particle. Global best objective value and local best objective values are initialized to 0. Global best particle position and local best particle position are empty initially.

After initialization, in the second step objective value of each particle is calculated based on Eq. (3) in a similar way as that calculated in the genetic algorithm. Local best value and global best value of fitness function are stored. The best particle with the best global value is also saved. In the next step, the velocity and position of each particle are updated based on Eqs. (4) and (5) [29].

$$v^i(t + 1) = w \times v^i(t) + c_1 r_1 (\text{best}_{\text{local}}^i - X^i(t)) + c_2 r_2 (\text{best}_{\text{global}} - X^i(t)) \tag{4}$$

$$X^i(t + 1) = X^i(t) + v^i(t + 1) \tag{5}$$

While calculating the new position of the particle, if the value of any element of the particle goes beyond the range of the code metrics, which is between 0 and 1 in our case, then reinitialize that element of the particle randomly between 0 and 1.

Global best objective value and particle are returned after a hundred iterations. The global best particle represents the optimal threshold values of the features used for software fault prediction. Algorithm 3 shows the overall steps of the PSO algorithm applied to select optimal threshold values.

## Results and Discussion

This section of the research article shows the results of the experiments performed on selected datasets. Experimentation is done on an HP Pavilion laptop with 8 GB RAM and a corei5 processor.

### Datasets

Table 2 shows the selected datasets for our experimentation. Datasets are downloaded from the PROMISE repository [27] and selected based on recent studies [30–34]. Datasets are divided into two categories in Table 2. The first ten datasets are created based on object-oriented code metrics, and the last four datasets are created based on procedural code metrics because

Algorithm 3: Pseudo code of PSO based threshold calculation

**Input:**

X: Training Samples  $X \in \mathbb{R}^{n \times m}$

y: Class labels  $y \in \{0, 1\}^n$

**Step 1:** Initialize twenty particles and their velocities randomly.

**Step 2:** calculate the objective value of each particle based on equation (3).

**Step 3:** Save the global best particle and its objective value.

**Step 4:** Save the local best objective value and local best particle.

**Step 5:** update the velocity of each particle based on equation (4).

**Step 6:** update the position of each particle based on equation (5).

**Step 7:** reinitialize particle if value cross the lower bound or upper bound.

**Step 8:** repeat step 2 to 7 hundred times.

**Step 9:** return global best objective value and particle.

Table 2 Datasets used to perform experiments

Datasets	Instances	Features	Fault percentage
Lucene-2.4	340	20	59.70
Jedit-4.0	306	20	24.50
Ant-1.6	351	20	26.21
Jedit-4.1	312	20	25.32
Ivy-1.4	241	20	6.63
Camel-1.4	872	20	16.62
Ivy-2.0	352	20	11.36
Ant-1.5	293	20	10.92
Poi-3.0	442	20	63.57
Lucene-2.2	247	20	58.29
Ar3	63	28	12.69
Ar4	107	28	18.69
Ar5	36	28	22.22
Ar6	101	28	14.85

the last four software are developed in a procedural programming language. Datasets of software developed in object-oriented programming language contain 20 features each, which are WMC, CBO, RFC, LOC, LCOM, LCOM3, DIT, NOC, CA, CE, MAX\_CC, AVG\_CC, NPM, MOA, MFA, DAM, CAM, IC, and AMC. Datasets of software developed in procedural programming language contains 28 code metrics which are total\_loc, blank\_loc, comment\_loc, code\_and\_comment\_loc, executable\_loc, unique\_operands, unique\_operators, total\_operands, total\_operators, halstead\_vocabulary, halstead\_length, halstead\_volume, halstead\_level, halstead\_difficulty, halstead\_effort, halstead\_error, halstead\_time, branch\_count, decision\_count, call\_pairs, condition\_count, multiple\_condition\_count, cyclomatic\_complexity, cyclomatic\_density, decision\_density, design\_complexity, design\_density, normalized\_cyclomatic\_complexity, formal\_parameters.

This research article uses only six features for datasets of software developed in object-oriented programming languages: WMC, LOC, CBO, RFC, DIT, and LCOM because these six features provide the best results based on threshold

values compared to datasets with the existing 20 features. For datasets of software developed in the procedural programming language, selected code metrics are halstead\_volume, cyclomatic\_complexity, cyclomatic\_density, decision\_density, design\_density, and normalized\_cyclomatic\_complexity. Each dataset is divided into a 67–33% ratio. 67% data is used to find the threshold values, and the remaining 33% is used to test the performance. Each experiment is performed ten times on each dataset, and averages and standard deviations are collected to compare the techniques.

### Performance Metrics

Accuracy, sensitivity, and specificity performance metrics are selected to compare the performance. Accuracy is used to show the overall performance of the model. Specificity and sensitivity check how well the proposed approach differentiates the classes in binary classification problems. Accuracy, sensitivity, and specificity can be calculated based on Eqs. (6), (7), and (8) [28].

$$\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN} \quad (6)$$

$$\text{Sensitivity} = \frac{TP}{TP + FN} \quad (7)$$

$$\text{Specificity} = \frac{TN}{TN + FP} \quad (8)$$

### Parameter Configuration

- (1) ROC: step size = 0.0001
- (2) GA: iterations = 100, population size = 20, crossover = binary, mutation rate = 0.1
- (3) PSO: iterations = 100,  $c_1 = c_2 = 1$ , number of particles = 20

### Performance Comparison

This section of the research article compares the proposed threshold based software fault prediction technique with an existing technique based on Accuracy, specificity, and sensitivity performance metric.

### Selection of Features

Selection of good features is important while predicting software faults based on metric threshold values; otherwise, prediction accuracy drops drastically.

Table 3 shows the accuracy comparison of different threshold selection techniques with all features and selected features. Selected features are WMC, CBO, RFC, LCOM, LOC, and DIT for object-oriented software and halstead\_volume, cyclomatic\_complexity, cyclomatic\_density, decision\_density, design\_density, and normalized\_cyclomatic\_complexity for procedural software. It is not necessary that for all datasets, the selected feature set will be the same. It may differ for different datasets depending upon the initial feature set and values of the features. But, we can select the optimal set of features based on the algorithm presented in "Normalization".

**Table 3** Accuracy comparison of selected features with all features

Datasets	ROC		GA		PSO	
	Selected	All	Selected	All	Selected	All
Lucene-2.4	<b>0.64336</b>	0.58359	<b>0.66106</b>	0.60796	<b>0.66017</b>	0.62300
Jedit-4.0	<b>0.54851</b>	0.34257	<b>0.69009</b>	0.44131	<b>0.70396</b>	0.44253
Ant-1.6	<b>0.44137</b>	0.28534	<b>0.76810</b>	0.58331	<b>0.74827</b>	0.58121
Jedit-4.1	<b>0.54951</b>	0.36796	<b>0.70388</b>	0.46944	<b>0.69902</b>	0.46578
Ivy-1.4	<b>0.34249</b>	0.18875	<b>0.69000</b>	0.48972	<b>0.63125</b>	0.48738
Camel-1.4	<b>0.375</b>	0.17187	<b>0.62256</b>	0.47765	<b>0.62013</b>	0.47143
Ivy-2.0	<b>0.45982</b>	0.14188	<b>0.74273</b>	0.54115	<b>0.73162</b>	0.54236
Ant-1.5	<b>0.33195</b>	0.16288	<b>0.81443</b>	0.56281	<b>0.81443</b>	0.55345
Poi-3.0	<b>0.68356</b>	0.57341	<b>0.75684</b>	0.65684	<b>0.73835</b>	0.64383
Lucene-2.2	<b>0.6</b>	0.57895	<b>0.56707</b>	0.55731	<b>0.53780</b>	0.56780
Ar3	<b>0.23809</b>	0.14285	<b>0.65238</b>	0.20952	<b>0.58095</b>	0.21904
Ar4	<b>0.31388</b>	0.19166	<b>0.725</b>	0.21944	<b>0.64166</b>	0.21388
Ar5	<b>0.37500</b>	0.23333	<b>0.72499</b>	0.30833	<b>0.65833</b>	0.29999
Ar6	<b>0.23823</b>	0.16470	<b>0.64117</b>	0.28235	<b>0.57941</b>	0.23823
Average	<b>0.43862</b>	0.29498	<b>0.69716</b>	0.45765	<b>0.66752</b>	0.45356

Boldface letters show the best results



**Table 4** Accuracy comparison of metaheuristic algorithms with ROC based threshold

Datasets	ROC	GA	PSO
Lucene-2.4	0.64336 ± 0.0319	<b>0.66106</b> ± 0.0349	0.66017 ± 0.0398
Jedit-4.0	0.54851 ± 0.0458	0.69009 ± 0.0554	<b>0.70396</b> ± 0.0382
Ant-1.6	0.44137 ± 0.0272	<b>0.76810</b> ± 0.0323	0.74827 ± 0.0423
Jedit-4.1	0.54951 ± 0.0421	<b>0.70388</b> ± 0.0462	0.69902 ± 0.0694
Ivy-1.4	0.34249 ± 0.1114	<b>0.69000</b> ± 0.0693	0.63125 ± 0.0859
Camel-1.4	0.375 ± 0.0195	<b>0.62256</b> ± 0.0407	0.62013 ± 0.0434
Ivy-2.0	0.45982 ± 0.0741	<b>0.74273</b> ± 0.0448	0.73162 ± 0.0544
Ant-1.5	0.33195 ± 0.0535	<b>0.81443</b> ± 0.0586	<b>0.81443</b> ± 0.0470
Poi-3.0	0.68356 ± 0.0304	<b>0.75684</b> ± 0.0254	0.73835 ± 0.0277
Lucene-2.2	<b>0.6</b> ± 0.0525	0.56707 ± 0.0460	0.53780 ± 0.0524
Ar3	0.23809 ± 0.0928	<b>0.65238</b> ± <b>0.0953</b>	0.58095 ± 0.0791
Ar4	0.31388 ± 0.0735	<b>0.725</b> ± <b>0.0601</b>	0.64166 ± 0.0997
Ar5	0.37500 ± 0.1003	<b>0.72499</b> ± <b>0.0916</b>	0.65833 ± 0.1145
Ar6	0.23823 ± 0.0517	<b>0.64117</b> ± <b>0.0506</b>	0.57941 ± 0.0921
Average	0.43862	<b>0.69716</b>	0.66752

Boldface letters show the best results

Prediction accuracy with all features is very low, as shown in Table 3. So, choosing good features to achieve acceptable results is super important. The best values are shown in boldface letters in Table 3.

In the case of ROC-based software fault prediction, selected features give approximately 14% improvement; in the case of GA-based software fault prediction, selected features give 24% improvement; and in the case of PSO-based software fault prediction, selected features give a 21% improvement in accuracy. However, prediction results based on the ROC curve and selected features are poor and below

the acceptable range of 0.6 [1]. In the case of GA and PSO, achieved performance is above the acceptable range, and GA gives 3% better results than PSO.

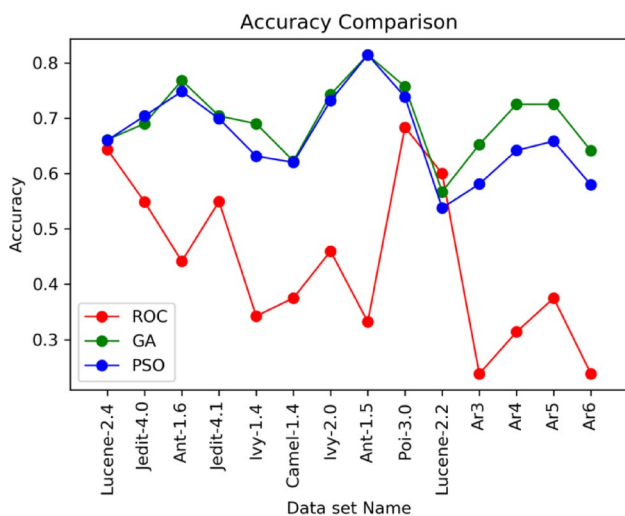
Based on Table 3, it can be concluded that selected features give a significant improvement in the performance of the threshold-based software fault prediction. Further, metaheuristic algorithms provide better relative threshold values of selected features than ROC, and in the case of metaheuristic, GA provides better results than PSO.

**Accuracy Comparison**

Table 4 presents the accuracy comparison of metaheuristic algorithms-based threshold detection technique with ROC-based threshold detection technique. The average accuracy and standard deviation with the ‘±’ sign of 10 runs are shown in Table 4. The best results are shown in boldface letters. GA gives the best results for 12/14 datasets. Out of the remaining two datasets, for Lucene-2.2, ROC gives the best results, and for Jedit-4.0, PSO gives the best results.

On average, ROC achieves 43% accuracy based on selected features that are poor to apply for real-life projects. On the other hand, metaheuristic-based selected threshold values achieve accuracy greater than 60%, which is acceptable. PSO-based thresholds achieve 66% average accuracy, which is not bad, and GA-based threshold values achieve 69% average accuracy, considered good and better 2% than PSO.

Figure 3 compares the accuracy of different threshold-based software fault prediction techniques. The red line shows the results of ROC-based threshold values, the Blue line shows the accuracy of PSO-based threshold values, and the green line shows the accuracy achieved by GA-based threshold values.



**Fig. 3** Accuracy comparison of ROC, GA, and PSO based threshold detection techniques

**Table 5** Sensitivity comparison

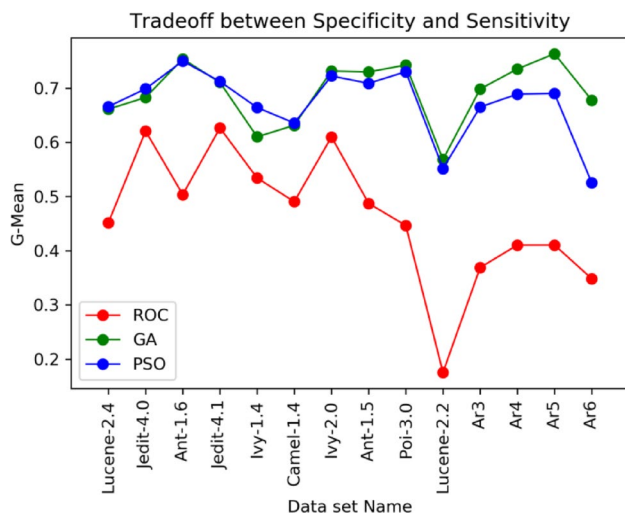
Datasets	ROC	GA	PSO
Lucene-2.4	<b>0.92853</b>	0.66298	0.63942
Jedit-4.0	<b>0.89383</b>	0.66744	0.68768
Ant-1.6	<b>0.91563</b>	0.72010	0.75500
Jedit-4.1	<b>0.90565</b>	0.72872	0.73511
Ivy-1.4	<b>0.93333</b>	0.53154	0.70142
Camel-1.4	<b>0.94373</b>	0.64654	0.65913
Ivy-2.0	<b>0.92896</b>	0.71605	0.70954
Ant-1.5	<b>0.89883</b>	0.63514	0.59432
Poi-3.0	<b>0.96649</b>	0.78911	0.76300
Lucene-2.2	<b>0.98872</b>	0.58419	0.51002
Ar3	<b>1.0</b>	0.76666	0.80666
Ar4	<b>0.97638</b>	0.75952	0.77186
Ar5	<b>1.0</b>	0.86833	0.775
Ar6	<b>0.95833</b>	0.72880	0.47027

Boldface letters show the best results

**Table 6** Specificity comparison

Datasets	ROC	GA	PSO
Lucene-2.4	0.21972	0.65932	<b>0.69360</b>
Jedit-4.0	0.43100	0.69827	<b>0.70978</b>
Ant-1.6	0.27694	<b>0.78905</b>	0.74479
Jedit-4.1	0.43324	<b>0.69278</b>	0.69006
Ivy-1.4	0.30563	<b>0.70008</b>	0.62882
Camel-1.4	0.25480	<b>0.61640</b>	0.61272
Ivy-2.0	0.40041	<b>0.74738</b>	0.73531
Ant-1.5	0.26417	0.83850	<b>0.84512</b>
Poi-3.0	0.20627	0.69872	<b>0.69887</b>
Lucene-2.2	0.03109	0.55273	<b>0.59656</b>
Ar3	0.13614	<b>0.63630</b>	0.54837
Ar4	0.17246	<b>0.71165</b>	0.61476
Ar5	0.16847	<b>0.67117</b>	0.61454
Ar6	0.12666	<b>0.63035</b>	0.58678

Boldface letters show the best results

**Fig. 4** Trade-off between sensitivity and specificity

The green line and blue line are almost overlapping each other. However, there is a slight difference between the green and blue lines, and we can say GA performs better than PSO because the green line is slightly above the blue line. Both GA and PSO outperform the ROC method, as shown in Fig. 3.

So, it is clear from Fig. 3 that the results of GA and PSO are comparable. However, GA gives slightly better results than PSO. The results of PSO and GA are far better than ROC based threshold detection technique.

## Other Metrics Comparison

Table 5 shows the sensitivity comparison of ROC, GA, and PSO-based threshold detection techniques over selected datasets. It is clear from Table 5 that ROC based threshold detection technique provides better sensitivity than the metaheuristic algorithms-based technique. Sensitivity, also called true positive rate, is better for ROC in the case of all datasets. But based on only a true positive rate, we can't say ROC-based threshold values develop a better model. We need to consider the true negative rate also.

Table 6 presents the comparison of the specificity of different threshold detection techniques. In the case of 9/14 datasets, GA gives better results, and PSO gives better results for the remaining five datasets. The metaheuristic algorithms-based technique for all ten datasets gives better results than the ROC-based threshold detection technique. However, based on only the true negative rate, we cannot conclude that the metaheuristic-based threshold provides a better software fault prediction model. We need to consider both the true positive and negative rates.

Figure 4 compares the specificity and sensitivity trade-offs of different threshold detection techniques. In the case of sensitivity, ROC gives better results, and in the case of specificity, the metaheuristic algorithms-based technique gives better results. However, metaheuristic algorithms give better results than ROC if we compare the sensitivity and specificity trade-offs. The results of different metaheuristic algorithms are comparable.

In Fig. 4, the red line shows the performance of ROC-based threshold values, the green line shows the performance of GA-based threshold values, and the blue line shows the performance of PSO-based threshold values. The blue and green lines are above the red lines in the graph. So, based on Fig. 4, we can conclude that metaheuristic-based threshold values are better than ROC for developing a good software fault prediction model.

## Threat to Validity

This study compares the metaheuristic-based threshold values with ROC-based threshold values for software fault prediction. We used fourteen open-source software, ten developed in object-oriented programming, and four developed in procedural programming. However, threshold-based fault prediction models' performance depends on features extracted from the code. We compare results based on only 20 object-oriented and 28 procedural code metrics, but many code metrics exist in the literature, and the model's performance can vary with changes in input features.

Second, we cannot say that the threshold values decided in this research article are universal because the experiment is done on a small set of datasets.

Third, thresholds decided by metaheuristic algorithms are not perfect. Metaheuristics always gives approximate solutions. However, performance is better than the ROC-based technique, and properly tuned algorithms can achieve good results.

## Conclusion

In this research article, we propose a new method based on metaheuristic algorithms to find the threshold values of software code metrics for software fault prediction. GA and PSO are used to find the threshold values, and performance is compared with ROC-based threshold values. In the first experiment, useful code metrics are selected based on experimentation out of available 20 object-oriented code metrics and 28 procedural code metrics because bad code metrics can drastically reduce the prediction performance of threshold-based software fault prediction models. In the second experiment, threshold values of selected code metrics are detected and used for software fault prediction. Accuracy, sensitivity, and specificity performance metric results are compared with ROC curve-based threshold detection techniques. Results show that metaheuristic algorithms give better threshold values for software fault prediction.

In the future, we will compare the performance of more metaheuristic algorithms for better and fast convergence and use threshold values for cross-project software fault prediction.

**Data Availability** Data will be made available on reasonable request to corresponding author.

## Declarations

**Conflict of Interest** The authors declare that they have no conflict of interest.

## References

- Boucher A, Badri M. Software metrics thresholds calculation techniques to predict fault proneness: an empirical comparison. *Inf Softw Technol.* 2018;96:38–67.
- Chidamber SR, Kemerer CF. A metrics suite for object oriented design. *IEEE Trans Softw Eng.* 1994;20(6):476–93.
- Shatnawi R, Li W, Swain J, Newman T. Finding software metrics threshold values using ROC curves. *J Softw Maint Evol.* 2010;22(1):1–16.
- Shatnawi R. A quantitative investigation of the acceptable risk levels of object oriented metrics in open-source systems. *IEEE Trans Softw Eng.* 2010;36(2):216–25.
- Gyimothy T, Ferenc R, Siket I. Empirical validation of object-oriented metrics on open source software for fault prediction. *IEEE Trans Softw Eng.* 2005;31(10):897–910.
- Malhotra R, Jain A. Fault prediction using statistical and machine learning methods for improving software quality. *J Inf Process Syst.* 2012;8(2):241–62.
- Jureczko M, Madeyski L. Towards identifying software project clusters with regard to defect prediction. In: *Proceedings of the 6th International Conference on Predictive Models in Software Engineering - PROMISE '10*, 2010. p. 1.
- Kaur A, Kaur K. Performance analysis of ensemble learning for predicting defects in open source software. In: *2014 International Conference on Advances in Computing, Communications and Informatics (ICACCI)*, 2014. pp. 219–225.
- Yu L. Using negative binomial regression analysis to predict software faults: a study of Apache ANT. *Int J Inf Technol Comput Sci.* 2012;4(8):63–70.
- Dejaeger K, Verbraken T, Baesens B. Toward comprehensible software fault prediction models using Bayesian network classifiers. *IEEE Trans Softw Eng.* 2013;39(2):237–57.
- Catal C, Sevim U, Diri B. Clustering and metrics thresholds based software fault prediction of unlabeled program modules. In: *ITNG 2009 - 6th International conference on information technology: new generations*, 2009. pp. 199–204.
- Abaei G, Selamat A, Fujita H. An empirical study based on semi-supervised hybrid self-organizing map for software fault prediction. *Knowl Based Syst.* 2014;74:28–39.
- Shatnawi R. Improving software fault-prediction for imbalanced data. In: *2012 International Conference on Innovations in Information Technology, IIT 2012*, 2012. pp. 54–59.
- Henderson-Sellers B. *Object-oriented metrics: measures of complexity*. Prentice-Hall, Inc; 1995.
- Daly J, Brooks A, Miller J, Roper M, Wood M. Evaluating inheritance depth on the maintainability of object-oriented software. *J Empir Softw Eng.* 1996;1(2):109–32.
- Cartwright M. An empirical view of inheritance. *Inf Softw Technol.* 1998;40(4):795–9.
- Emam K, Benlarbi S, Goel N, Rai S. The confounding effect of class size on the validity of object-oriented metrics. *IEEE Trans Softw Eng.* 2001;27(7):630–48.
- El Emam K, Benlarbi S, Goel N, Melo W, Lounis H, Rai S. The optimal class size for object-oriented software. *IEEE Trans Softw Eng.* 2002;28(5):494–509.
- Erni K, Lewerentz C. Applying design-metrics to object-oriented frameworks. In: *Proceedings of the third international symposium on software metrics: from measurement to empirical results*, 1996; 64–74.
- Bender R. Quantitative risk assessment in epidemiological studies investigating threshold effects. *Biom J.* 1999;41(3):305–19.
- Alves TL, Ypma C, Visser J. Deriving metric thresholds from benchmark data. In: *2010 IEEE International Conference on Software Maintenance*, 2010. pp. 1–10.
- McCabe T. A complexity measure. *IEEE Trans Softw Eng.* 1976;SE-2(4):308–20.
- Rosenberg LH (1998) Applying and interpreting object oriented metrics. In: *Software Technology Conference*.
- Singh S, Kahlon KS. Object oriented software metrics threshold values at quantitative acceptable risk level. *Csit.* 2014;2(3):191–205.
- Benlarbi S, El Emam K, Goel N, Rai S. Thresholds for object-oriented measures. In: *Proceedings 11th International Symposium on Software Reliability Engineering. ISSRE 2000*, IEEE Comput. Soc, 2000. pp. 24–38

26. Catal C, Alan O, Balkan K. Class noise detection based on software metrics and ROC curves. *Inf Sci.* 2011;181(21):4867–77.
27. Boetticher G. The PROMISE repository of empirical software engineering data, 2007. <https://cir.nii.ac.jp/all?q=http://promisedata.org/repository>
28. Canbek G, Sagioglu S, Temizel TT, Baykal N. Binary classification performance measures/metrics: A comprehensive visualized roadmap to gain new insights. In: 2017 International Conference on Computer Science and Engineering (UBMK), IEEE, 2017. pp. 821–826.
29. Kennedy J, Eberhart R. Particle swarm optimization. In: Proceedings of ICNN'95-international conference on neural networks, vol. 4. IEEE, 1995. pp. 1942–1948.
30. Rathi SC, Misra S, Colomo-Palacios R, Adarsh R, Neti LBM, Kumar L. Empirical evaluation of the performance of data sampling and feature selection techniques for software fault prediction. *Expert Syst Appl.* 2023;223: 119806.
31. Sharma U, Sadam R. How far does the predictive decision impact the software project? The cost, service time, and failure analysis from a cross-project defect prediction model. *J Syst Softw.* 2023;195: 111522.
32. Feng S, Keung J, Zhang P, Xiao Y, Zhang M. The impact of the distance metric and measure on SMOTE-based techniques in software defect prediction. *Inf Softw Technol.* 2022;142: 106742.
33. Arar ÖF, Ayan K. Deriving thresholds of software metrics to predict faults on open source software: replicated case studies. *Expert Syst Appl.* 2016;61:106–21.
34. Nevendra M, Singh P. Empirical investigation of hyperparameter optimization for software defect count prediction. *Expert Syst Appl.* 2022;191: 116217.

**Publisher's Note** Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Springer Nature or its licensor (e.g. a society or other partner) holds exclusive rights to this article under a publishing agreement with the author(s) or other rightsholder(s); author self-archiving of the accepted manuscript version of this article is solely governed by the terms of such publishing agreement and applicable law.