Turkish Online Journal of Qualitative Inquiry (TOJQI)
Volume 12, Issue 6, June 2021: 703-713

Certain Exploration of Code Smell Empathy and Refinement Employing Random Decision Forest Classifier

M.Sangeetha [a*], Dr.C.Chandrasekar [b]

[a*] Research Scholar, Department of Computer Science, Periyar University, Salem-11
[b] Professor, Department of Computer Science, Periyar University, Salem-11

***Corresponding author:** mslion2010@gmail.com, ccsekar@gmail.com

Abstract

All through the changes in the source code there might be anmanifestation of smell. Fowler calls as a Bad Smell which is the symbols of potential problem in the code that may necessitate a refactoring. Inexperienced software felt challenging to resolve the bad smells even with the help of refactoring tools. The intention is that the developers don't know where to invoke the refactoring tools and how to choose the refactoring tools for identifying various code smells. For this projected a framework to perform refactoring promptly. By this the developer may refactor the bad smells and resolves them promptly. Refactoring is a well-established practice that aims at improving the internal structure of a software system without changing its external behavior. Existing literature provides evidence of how and why developers perform refactoring in practice. In this proposed work to continue on this line of research by support the MICA-GRDFC technique is more capable and operative than state of the art refactoring techniques practices in 200 open source systems..

Keywords: Software code smell, Smell identification, Smell rectification, Random tree forest classifier, Cost optimization

1. Introduction

In contemporary days, there are number of automated tools and methods obtainable for detected code smells in exercise. Developers or software engineers are little bit potential to handle these automated tools. Refactoring tools are used to improve the performance of software. Code smells are the chaotic problem in software system which condenses the software quality. A monitor-based instant refactoring structure was introduced for detecting and avoidsquantity of code smells in source code. However, the framework has enormousconservation cost and miscarried to recover software quality. A parallel Evolutionary algorithm (P-EA) was developed to detect code-smells popular software system. However, the automatic correction of code-smells persistednot considering now. A flexible and lightweight technique was presented for detecting and showing the code smells from multiple software languages. But, this technique does not detect dissimilar types of code smells. Several linear regression analyses were accomplished for examining the association between codes smells as well as code size with less conservation efforts. However, it failed to focus on software code size for obtaining less maintenance effort.An effective clone detection technique was introduced using three different tools and examines the refactoring on various software quality metrics. But, the method was not analyzed several code smells on the system and check their occurrences before and after refactoring. A new algorithm was designed for automated detection of refactoring process to the strategy design model. But,the false positive rate was not reduced during the refactoring. A genetic algorithm-based approach was introduced for refactoring the code smell of component-based software. But, it failed to use multi-objective optimizations to improve the

performance of refactoring. A reliable and efficient method was developed for automatically evaluate the software clones refactoring. However, it has high computation cost for refactoring. A new technique was introduced for refactoring the package construction of object oriented software. However, it failed to use more refactoring technique for rectify the code smells in source code. An automated technique was developed for detecting refactoring with cost effective in object-oriented software. However, it failed to consider the different types of code smell and it was not efficient to select the best refactoring to replace code smell. The certain issues are identified from above said existing methods such as, high false positive rate, computation cost and lack of code-smells rectification, failed to select best refactoring technique, more effort to maintain software code and so on. This proposal work introduced Multivariate independent component analysis and generalized random decision forest classifier technique (MICA-GRDFC). MICA-GRDFC creates two contribution of our proposal. First Multivariate independent component analysis is measured mutual dependence between lines of codes and rules of code smells. The requirement measure is used for finding the association or correlation between multiple code lines and rules. Based on dependence measure, developer identifies which types of code smell are presented in that code lines. This helps to improve software code smell identification accuracy with a reduced amount of false positive rate. Second generalized random decisionforest classifier is proposed for code smell rectification in source code lines. GRDFC constructs number of decision tree with randomly selected training samples (i.e. refactoring technique). All the decision trees are combined and applied majority ballotingsystem.

Themajority votes are selected for rectifying the code smells in source code lines. This in prospectdevelops the code smell renovation in terms of true positive rate with less computation cost. The statistical analysis of the obtained results provides evidence to support the MICA-GRDFC technique is more efficient and effective than state of the art refactoring techniques. A Multivariate independent component analysis based generalized random decision forest classifier (MICA-GRDFC) technique is introduced for performing both code smell types identification and smell renovation with less computation cost. The processing flow diagram of MICA-GRDFC techniques is shown in figure 1.

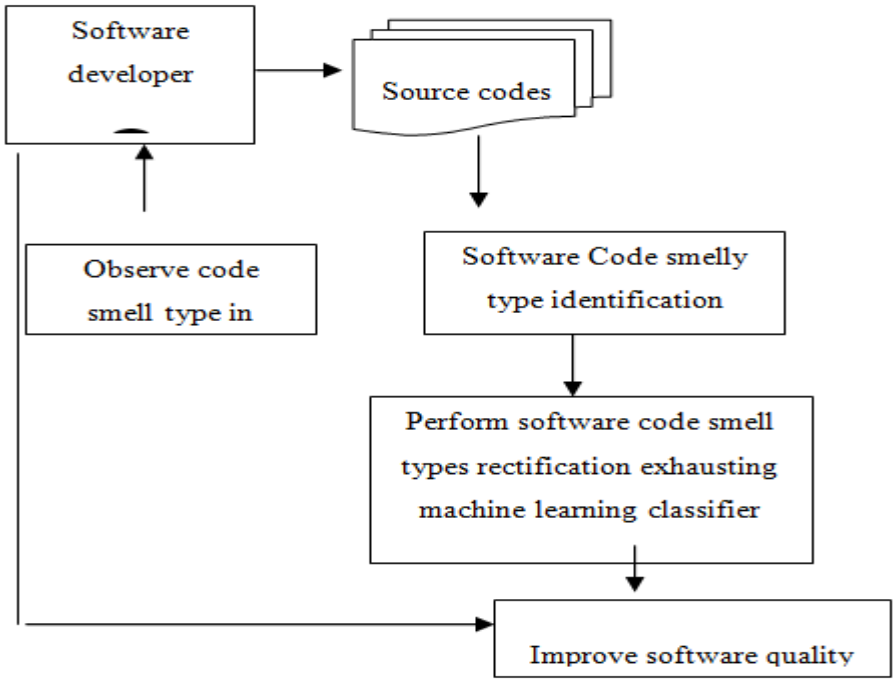

Figure 1 Flow processing diagram of Multivariate independent component analysis

Multivariate Independent component analysis (MICA) is a machine learning method for identifying a code smell types in source code. The normal independent component analysis acts as a linear transformation considering the mutual statistical independence of the non-Gaussian source signals. But it does not considered a multiple variables to forecast possible outcomes. In addition, MICA method often directs to local minimum solution which difficult to detect exact code smells in source code. Therefore, (i.e. sum of code smells in source code) and provides global optimum solutions in the code smell identification. The MICA-GRDFC technique used for transforming observed multivariate data (i.e. source code) into statistically autonomous components which are suggested as linear combinations of observed variables.In general, code smells in computer programs refers to any indication in the source code of a program that possibly creates a deeper problem. There are

different types of codes smells are presented namely application level, class level and method level in source code. Multivariate Independent component analysis correctly identifies which types of code smell are presented in source code based on the certain code rules. The certain rules of code smells are similar code exists, many instance variables, class having little functions, redundant code and so on. Based on above said rules, the code smell types are classified using Multivariate Independent component analysis. If source code line has the rule of code smell, then the developer identified which types of code smell is presented. Flow processing diagram of multivariate independent component analysis based software code smell type's identification is shown in figure 2.

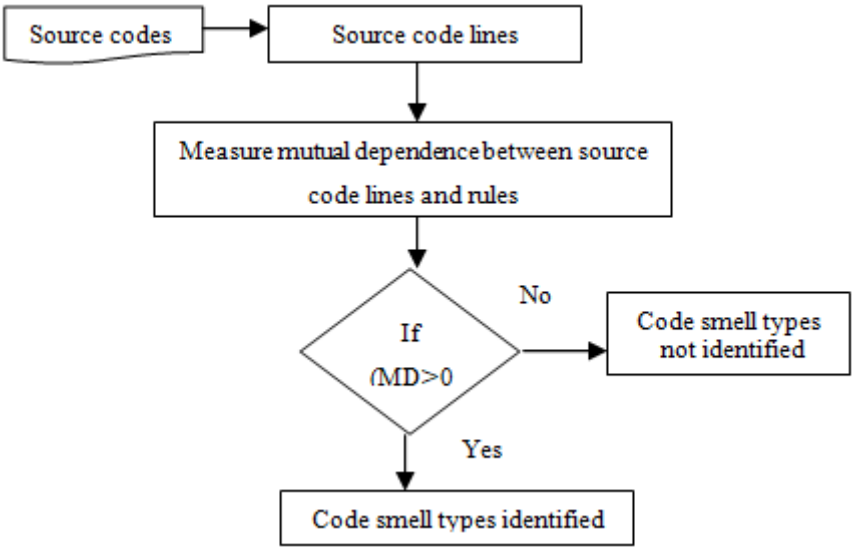

Figure 2 flow process of multivariate independent component analysis

Figure 2 shows the flow processing diagram of multivariate independent component analysis to identify the different type's code smell in source code lines. The input of source codes containsa number of source code lines. The certain predefined rules are used to identify the code smellscategories in source code lines. The relationship between source code and the rules of code smell are denoted as follows,

$S_i = BR_i$---------- (1) i ∈ 0,1,2 … n

From (1)'S_i' indicates a number of 'm' dimensional source codes and it contains a number of lines $S \in l_1, l_2 \ldots l_n$ and 'R_i' represents a 'n' dimensional independent components (i.e. certain rule) $r_1, r_2, \ldots . r_n$. Bdenotes a constant mxn mixing matrix. The aim of multivariate independent component analysis is to identify the smell from the measured data matrix (i.e. source code). By improving the multivariate independent component analysis, mutual information between 'n' variables (i.e. multivariate) is minimized. The mutual information is a measure of mutual dependence between the source code lines and certain rules. The input of source code contains a number of lines. MICA verifies each lines of source code to provide the best possible solutions of code smell out of all possible solutions (i.e. global optimum solutions). If the source code line has the rules of code smell, then the types of code smells are identified. Therefore, mutual dependence is described as,

$$MD = \sum_{l \in S} \sum_{r \in R} p(l,r) \log_{10} \left(\frac{p(l,r)}{p(l),p(r)} \right) \text{----- (2)}$$

From (2), where $p(l,r)$ denotes a joint probability distribution of source code lines and rules that gives the classification probability that falls in any source code. $p(l)$and$p(r)$ represents a marginal probability of source code lines and rules of code smell. Mutual dependence is a measure of the intrinsic dependence expressed in joint distribution of *'l'* and *r* under the assumption of independence. If *'l'* and 'r' are independent then the joint probability is described as,

$$p(l,r) = p(l) * p(r) \text{--------(3)}$$

By substituting equation (3) in (2), mutual dependency is derived as follows,

$$MD = \sum_{l \in S} \sum_{r \in R} p(l) * p(r) \log_{10} \left(\frac{p(l) * p(r)}{p(l) * p(r)} \right) \text{---- (4)}$$

From the above equations are solved by,

$MD(l,r) = \sum_{l \in S} \sum_{r \in R} p(l) * p(r) \log_{10}(1)$------- (5)

$MD(l,r) = 0$----- (6)

From (5), the value of log 1 is zero and it is shown in equation (6). Therefore, it shows independence between code lines and rules of code smells. Mutual independence is measured and it expressed in joint distribution of *'l'* and *r* under the assumption of dependence. If *'l'* and 'r' are dependent then the joint probability is described as,

$p(l,r) \neq p(l) * p(r)$----- (7)

Therefore, the mutual dependence is obtained by,

$MD(l,r) = \sum_{l \in S} \sum_{r \in R} p(l,r) \log_{10} \left(\frac{p(l,r)}{p(l),p(r)} \right)$, $MD(l,r) \geq 0$----------(8)

Therefore,

$$MD(l,r) = \begin{cases} 0 & l,r \; independent \\ \geq 0 , & l,r \; are \; dependent \end{cases}$$ ---------- (9)

Therefore, if $MD(l,r) \geq 0$, provides the correlation between the source code lines and rules. it is identified as a certain type of code smells presented in a source code lines. Mutual dependence measured association or correlation between the multiple variables (i.e. multiple code lines and rules) by evaluating both dependence and independence of source code lines and rules. This provides the global optimum solution by verifying each lines of source code to detect code smells.Algorithmic description of Multivariate independent component analysis is shown inbelow

Input: Source code', source code lines $l_1, l_2 \ldots l_n$, certain rules $r_1, r_2, \ldots . r_n$

Output: improve software code smells type identification accuracy

Step 1: Create

Step 2: For every source code lines

Step 3: Measure relationship between source code and the rules using (1)

Step 4: Measure mutual dependence between 'l' and 'r' using (4)

Step 5: Measure mutual independence between source code lines and rule of code smell using (8)

Step 6: if (MD==0) then

Step 7: Software code smells type is not identified

Step 8:endif

Step 9: if (MD≥0) then

Step 10: Software code smells type is identified in source code lines

Step 11::endif

Step 12: endfor

Step 13: End

Initially, the mutual relationship between the source codes lines and the rules are measured. Then the dependence between the source codes lines and the rules provides the correlation and it effectively used for identifying the software code smells types in source code. This supports to improve the smell identification accuracy with least false positive rate.

2. Generalized random decision forest classifier for code smells type rectification

When the code smells types are recognized, the MICA-GRDFC technique performs code smells type rectification using generalized random decision forest classifier. When reporting code smell identification results, software developer refactors the code smell types in source code lines with minimum cost utilization. The code smells in source code lines are particularly large and complex, and generates larger issues to the maintainability of software system. Therefore, a machine learning classifier namely generalized random decision forest classifier technique is applied for efficient rectification. The limitation of general Random forests classifier takes more inputs samples and its slow to evaluate and provides misclassifications. But our generalized random decision forest classifier calculates error model during the classification to avoid the slow process and improves the classification performance. GRDFC rectify the code smell types without altering the internal behavior of the source codes. This in turn improves the code quality and maintainability of software system. Source code design and its quality are boosted by applying refactoring thus increases the code reusability. Generalized random decision forest classifier has 'n' number of binary decision trees. From the below figure, each decision trees are qualified by a random approach. GRDFC selects random samples correlated to the identified code smells from collections of training samples (i.e. refactoring techniques). The final decision combines all the outputs from the individual decision tree.

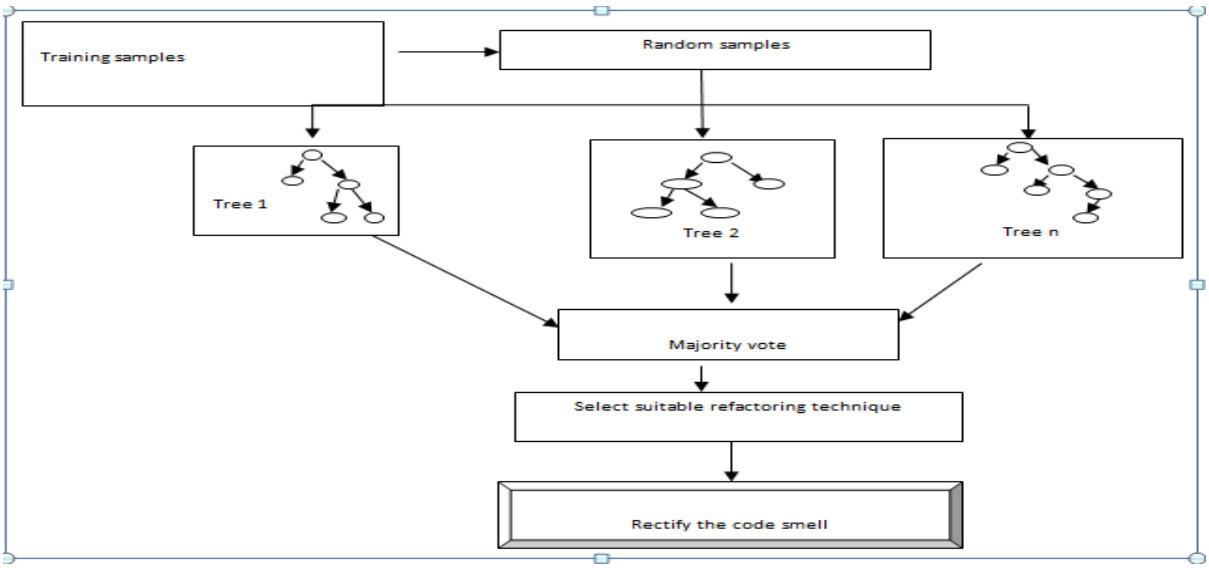

Figure 4 process of generalized random decision forest classifier

Let us consider, GRDFC takes the refactoring techniques as training samples $(s_1, y_1), (s_2, y_2) \ldots (s_n, y_n)$ to perform efficient code smell rectification. From the number of training samples, $(s_1, s_2 \ldots, s_n)$ a random sample with replacement of the training set and fits trees to these sets, and $y_1, y_2, .. y_n$ represents output of class labels. GRDFC rectifies the code smells in source code by exchanging the suitable refactoring technique (i.e. samples). GRDFC randomly constructs a decision tree with the training samples and classifies which refactoring technique is more suitable for rectifying the particular type of code smells through majority vote. The majority vote of samples is predicted from the trees for rectification.

The decision tree is mostly employed in classification as it is fast and provides more efficient results. Based on given input training samples, a decision tree creates a model that predicts the value of a target variable (i.e. refactoring technique). Each internal node in tree denotes a test on training samples. A leaf node holds the different classes and delivers an effective prediction of refactoring technique to interchange the code smell in source code. The output of the decision tree is combined as follows,

$$\hat{y} = f(x_1) + f(x_2) + \cdots + f(x_n) - - - (10)$$

From (10),$f(x)$ denotes an output of individual tree classifier. After training, predictions of training samples are obtained by averaging the predictions from all the individual decision trees. Therefore, the output of generalized random decision forest classifier is expressed by applying the majority vote,

$$\hat{y} = V\{f(x_1) + f(x_2) + \cdots + f(x_n)\} - - - (11)$$

From (11), V represents a vote applied for number of samples in individual tree classifier$f(x)$. Therefore output of classifier is formulated as,

$$\hat{y} = arg \max_{n} \quad \{s|s \in s_n\}\text{-----} \quad (12)$$

From (11),$\hat{y}$ denotes a final predicted classifier output with majority votes and s denotes a suitable refactoring technique whose decisions are known to the n^{th} classifier. The arguments of the maxima (arg max) are the maximum value of some function. Therefore, a suitable refactoring technique is predicted from number of samples(s_n). During the classification, the generalized random decision forest classifier minimizes the generalization error to improve the classification. In MICA-GRDFC technique, generalization error is a measure of how accurately an algorithm is effectively predicts the outcomes. Therefore, the generalization error is defined as the difference between the predictable and observed error. This is the difference between error on the training set and error on the fundamental joint probability distribution. It is measured as follows,

$$E_g = E_p - E_o\text{-------} \quad (13)$$

From (13),E_g denotes a generalization error, E_p and E_o denotes an error of predicted and observed error respectively.

$$E_p = \sum_{i=1}^{n} L(f(x_i), \hat{y})\, p(s_i, \hat{y}) - - - \quad (14)$$

From (14), where $L\, f(x_i)$ denotes a loss function of individual classifier and $p(s_i, \hat{y})$ represents an unknown joint probability distribution for the input of training samples and y represents a predicted output values. Observed error is measured as follows,

$$E_o = \frac{1}{N}\sum_{i=1}^{n} L(f(x_i), \hat{y})\text{---} \quad (15)$$

From (15),N denotes a number of input training samples. MICA-GRDFC technique measures the generalization error and it reduced for improving the accuracy of rectification with minimum computation cost. Initially, training samples are considered for perform code smell type rectification. From the training samples, the random samples are selected which are relevant to the detected code smells types in source code lines. Then these samples are trained with decision tree classifier. These classifiers are combined into one classifier and applied voting for achieving the prediction of decision. Therefore the majority vote of the random set (i.e. refactoring technique) is predicted in the decision of n^{th} class. During the taxonomy, the generalization error is measured for avoiding the misclassification. Therefore, it helps to select the appropriate refactoring technique and correct the code smells in source code lines.

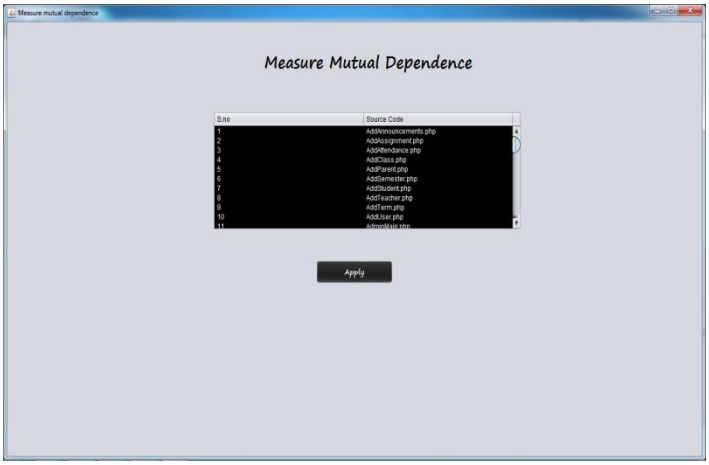

Figure 4. Measured Mutual Dependence

Figure 5. Rectification code smell using GRDFC

3. Research Questions

This proposed technique aims to drive inexperience software engineers to apply more refactoring promptly. The goals include two aspects.

(1) Identify more code smell

(2) Improve true positive rate

As a decision, the initial estimation should examine the following research questions

RQ1: Which kinds of refactoring technique practical software system such as Android, Apache and Eclipse?

RQ2: How far of precision to recognize the code smells in source code?

4. Experimental Settings

Our training samples takes 500 open source projects and well recognized ecosystems Android, Apache and Eclipse.Software developer monitors every lines in the source program to observe the codes smell and also identify which types of code smells are existing in particular line. The MICA-GRDFC technique performs efficient code smell types identification and rectification in source code lines.The experimentation is accompanied with the parameters such as software code smell type's identification accuracy, false positive rate, true positive rate and computation cost with respect to input source code.

Table1: Sample open source projects

Ecosystem	#Proj	#Classes	KLOC
Apache	100	4-5052	1-1031
Android	70	5-4960	3-1140
Eclipse	30	142-16,700	26-2610

5. Results And Discussion

Result and discussion of MICA-GRDFC techniques are explained and compared with existing monitor-based instant refactoring framework and P-EA approach. The presentation analysis is approved out with number of factors such as software code smell type identification accuracy, false positive rate, true positive rate and computation cost with respect to input source code. The results are discussed with the help of tables and graph values

5.1 Impression of software code smell types identification accuracy

Software code smell types Identification accuracy is measured constructed on the number of code smells type are identified appropriately in source code. The source code has number of lines. The software code smell type identification accuracy is measured as follows

$$SCMTIA = \frac{N_{sl} - code\ smell\ type\ is\ correctly\ identified}{N_{sl}} * 100 \quad (16)$$

From (16), Where $SCMTIA$represents a software code smell type identification accuracy and 'N_{sl}' represents a number of source code lines. It is measured in the unit of percentage (%).

Table 2: Software code smell type Identification accuracy

Source code (KB)	**Software code smells type Identification accuracy (%)**		
	MICA-GRDFC	**Monitor-based instant refactoring framework**	**P-EA approach**
2	86	62	73
4	87	65	75
6	88	69	78
8	90	70	82
10	91	72	83
12	93	75	84
14	94	78	85
16	95	80	86
18	96	82	88
20	97	84	89

As shown in figure 6, performance results of software code smell type identification accuracy is illustrated. The below figure clearly illustrations that the accuracy is significantly improved than the existing methods. The existing monitor-based instant refactoring framework performs code smells detection. But the rule based code smell detection was not performed. On the contrary, the proposed MICA-GRDFC technique uses multivariate independent component exploration identifies the code smell type in source code lines by applying certain rules of code smell.

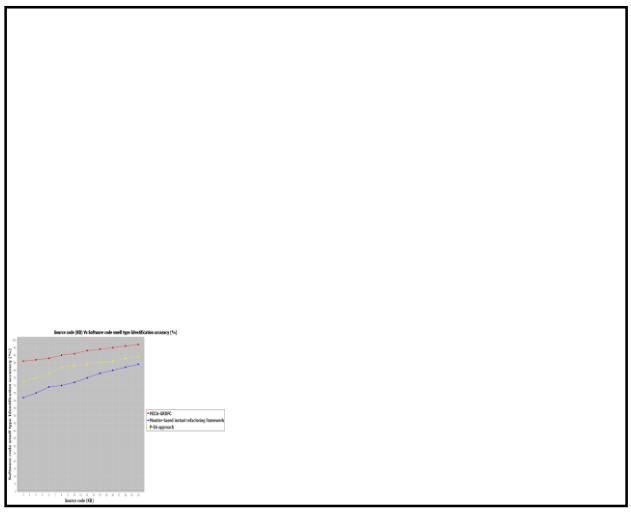

Figure 6 performance results of software code smell type identification accuracy

5.2 Impact of false positive rates

False positive ratesare measured based on proportion of number of code smell types are inaccurately identified to the number of lines in source code. The false positive rate is measured as follows,

$$FPR = \frac{No.of\ code\ smell\ types\ is\ incorrectly\ identified}{N_{sl}} * 100 \quad (17)$$

From (17), where FPR represents false positive rate and 'N_{sl}' denotes a number of source code lines in software code. It is measured in terms of percentage (%).

Table 3 Tabulation for false positive rate

Source code (KB)	**False positive rate (%)**		
	MICA-GRDFC	**Monitor-based instant refactoring framework**	**P-EA approach**
2	20	42	30
4	23	44	32
6	25	45	33
8	26	47	34
10	29	50	37
12	30	52	39
14	32	53	40
16	34	55	41
18	36	57	43
20	38	58	45

Performance evaluation of false positive rate with respect to source code is described. For the experimental evaluation, the input of source code is taken from 2KB to 20KB. Each source code has number of lines. From the table value, MICA-GRDFC technique reduces the incorrect detection of code smell types in source code lines when compared to existing monitor-based instant refactoring outline and P-EA method respectively.

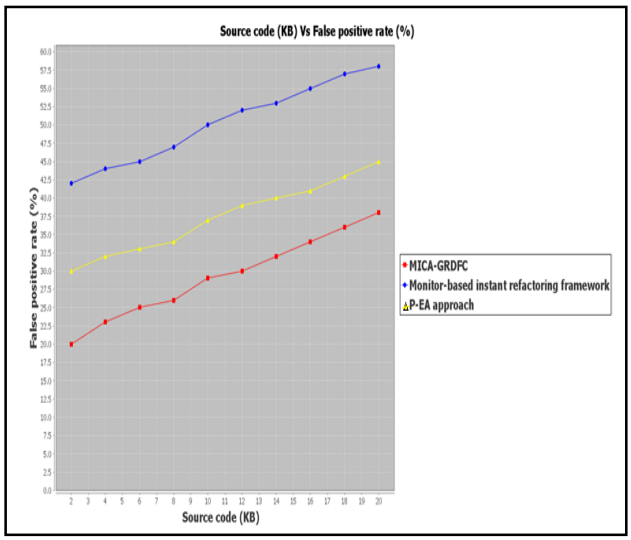

Figure 7: Performance results of false positive rate

5.3 Impact of computation cost

Computation cost is measured based on amount of time required for rectifying the code smell types in source code program. The formula for computation cost is measured as follows.

$$CC = no.of\ code\ smells * T(rectify\ the\ code\ smell\ type) \quad (19)$$

From (19), where CC denotes computation cost, $'T'$ represents time for the code smell types rectification in a source code lines. It is measured in terms of millisecond (ms).

Table4. Tabulation for computation cost

Source code (KB)	Computation cost (ms)		
	MICA-GRDFC	Monitor-based instant refactoring framework	P-EA approach
2	8	20	13
4	10	30	16
6	16	42	28
8	23	54	36
10	30	63	48
12	38	79	56
14	43	87	63
16	57	107	78
18	66	117	93
20	78	128	97

Experimental results of computation cost with respect to source code are described in table 4. The table values shows that the calculation cost is reduced using MICA-GRDFC technique when compared to existing using monitor-based instant refactoring framework [1] and P-EA approach [2]. This is due to the projectedMICA-GRDFC technique uses generalized random forest decision classifier. The comparison results of computation cost are shown in figure 9.

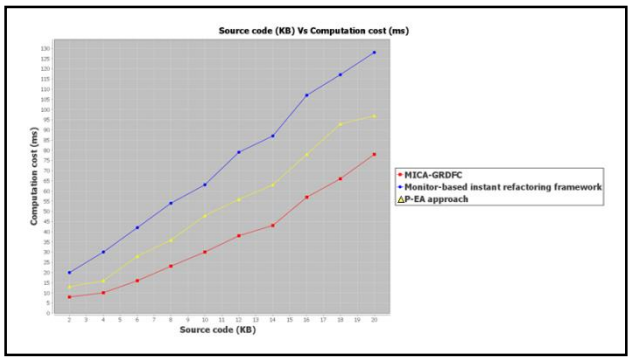

Figure 9 performance results of computation cost

From the observations, the computation cost is suggestively reduced by 53% and 33% when related to monitor-based instant refactoring framework [1] and P-EA approach [2] respectively. From the above said discussions, MICA-GRDFC technique identified and rectified the code smells and its types by using suitable refactoring technique with less computation cost.

6 Conclusion

An well-organized Multivariate independent component investigation based generalized random decision forest classifier (MICA-GRDFC) machine learning technique is obtainable for achieving value assured software refactoring. At first, the software developer recognizes number of code smells and its types in a source code. The code smell identification is proficient using multivariate independent component analysis. The MICA performs mutual dependence and independence between source code lines and certain rules of code smells. The dependence between source code lines and rules of code smells are used to recognize which types of code smellsare presented in source code. This in turn develops code smell type identification accuracy with less false positive rate. Furthermore, the developer performs code smells rectification by applying generalized random decision forest classifier. The GRDFC execute efficient classification by constructing the binary decision tree with the number of random samples related with code smells. The maximum polling is applied for aggregating all the decision tree classifier. Then the best refactoring technique is forecast and it is used for code smell rectification with less computation cost and high true positive rate. Experimental evaluation is achieved using schoolmate dataset with the parameters are software code smell type identification accuracy, false positive rate, true positive rate and computation cost. The results analysis of MICA-GRDFC technique progressessoftware code smells type identification accuracy and true positive rate with less computation time as well as false positive rate than the state-of-art methods.