International Journal of Innovative Technology and Exploring Engineering (IJITEE)
ISSN: 2278-3075 (Online), Volume-9 Issue-4, February 2020

Engrossing Prosecution of Code Smells Type Identification and Rectification using Machine Learning AdaBoost Classifier

M. Sangeetha, C. Chandrasekar

Abstract*: Software code smells are the structural features which reside in a software source code. Code smell detection is an established method to discover the problems in source code and reorganize the inner structure of object-oriented software for improving the quality of such software, particularly in terms of maintainability, reusability and cost minimization. The developer identified where the code smell is identified and rectified within a system is a major challenging issue. The various code smell detection technique has been designed but it failed to classify the code type and minimum rectification cost. In order to perform classification with minimum cost, an efficient technique called Machine Learning Ada-Boost Classifier (MLABC) technique is introduced. The MLABC technique improves the software quality by identifying and rectifying the different types of software code smell in source code. Initially, MLABC technique uses decision tree as base classifier to identify the code smell type. The decision tree is used to classify the code smell type based on the certain rule. After that, the base classifiers are combined to make a strong classifier using adaboost machine learning technique. The output of strong classifier is used to identify the code smell type. Finally, the code smell type rectification is performed by applying the refactoring technique where the code smell is identified with minimum cost and space complexity. Experimental results shows that the proposed MLABC technique improves the software code quality in terms of code smell type identification accuracy, false positive rate, code smell type rectification cost and space complexity with the source code.

***Keywords*: Cross-cultural projects, Human-computer interface, learning communities, code smell type identification, code smell type rectification.**

I. INTRODUCTION

In software programs, a code smell creates a deeper problem in the source code to degrade the software quality. In general, such kind of problem in the code is called as code smell and the detection of code smells has become a well-known method to identify the software design issues that may cause problems for further maintenance.

Revised Manuscript Received on February 28, 2020.
***** Correspondence Author
M. Sangeetha*, Ph.D Research scholar, Department of computer science, Periyar University, Salem -11 INDIA (e-mail: mslion2010@gmail.com).
Dr. C.Chandrasekar, Professor, Department of computer science, Periyar University ,Salem -11 INDIA, (ccsekar@gmail.com)

Therefore, these smells are rectified to avoid the problems. But the costs and risks are major parameters for rectifying the code smell. Many techniques have developed to support the detection of code smells. In [1], a monitor-based instant refactoring framework was developed to determine and avoid more code smells rapidly. However, it has high maintenance cost and failed to improve software quality. A dynamic and Automatic Feedback-Based Threshold Adaptation technique was introduced in [2] to detect the code smell in source code. However, the false positive was not reduced to improve the software equality.

An automated approach was developed in [3] for detecting the cost-effective refactoring with the help of dynamic information in object-oriented software. However, it failed to consider the different types of code smell in software program. Harmfulness code smell model was developed in [4] for identifying and classifying the harmful effect of code smells. But the other types of code smell were not identified.

The analysis of four code smells were presented in [5] throughout consecutive versions of two open source system. However the experimental analysis on many software systems and other code smells were not validated. A semi-automated approach was developed in [6] for prioritizing code smells before selecting on suitable refactoring. But, the refactoring cost of such method was high. In [7], multiple linear regressions analysis in which the entire code smells were investigated in the similar model. However, it reduced the software maintenance. Six types of bad smells were detected in [8] which were rectified by using window based graphical user interface. But, it failed to use different metrics to detect more code smells. The different code smell detection tools were compared in [9] by analyzing their accuracy from the reference list. However it failed to achieve software maintenance performances. Parallel Evolutionary algorithm (P-EA) was designed in [10] with a similar cooperative manner for detecting the code-smells. However, the automatic rectification of code-smells was not performed. The certain issues are identified from above said existing methods such as high cost, high false positive rate, failed to perform different types of code smell identification, failed to perform code rectification with minimum cost and so on. In order to overcome such kind of issues, Machine Learning Ada-Boost Classifier (MLABC) technique is introduced. Contribution of the paper is described as follows,

- Machine Learning Ada-Boost Classifier (MLABC) technique is developed to improve software quality and reusability. AdaBoost with decision tree classifier is designed to identify the types of code smells are occurred in source code program. Base decision tree classifier performs effective classification using specific rule related to the code smell. The base classifiers are combined to create strong classifier based on weight value. The output of strong classifier is used to improve code smell type identification accuracy with less false positive rate.
- The refactoring technique is applied for rectifying the code smell types in source code. This helps to change the internal structure of source code and it is not affected the external structure. The rectification process is used to remove the duplication type of code and other unused code in source code programs. This helps to reduce the code smell rectification cost and space complexity.

The paper is arranged with following sections. In Section 2, Machine Learning Ada-Boost Classifier (MLABC) technique is explained with neat diagram. In Section 3, Experimental evaluation is presented and the simulation results are discussed in section 4. In Section 5, related works are reviewed and discussed briefly. Finally, the conclusion of research work is presented in section 6.

II. MACHINE LEARNING ADA-BOOST CLASSIFIER FOR SOFTWARE CODE SMELL TYPE IDENTIFICATION AND RECTIFICATION

In software programming code, codes smell is a bad smell that affects software quality by creating a significant problem. Bad smells are fundamental problems in a source code that the developer difficult to maintain the software programs effectively. This bad smell creates faults in the source code which results software getting slower down or risk of error is increased. Therefore, if any bad smells in source code are rectified for improving the software quality. Rectification is used for correcting the bad smells from source code of software without affecting the external behavior of code. Therefore, code smell identification and rectification are performed by using efficient technique. With this motivation, Machine Learning Ada-Boost Classifier (MLABC) technique is introduced to improve software code quality with minimum cost. The processing diagram of ML-ABC technique is described as shown in figure 1.

Figure 1 shows the processing diagram of ML-ABC technique for code smell type identification and rectification to improve the software quality with minimum cost. The ML-ABC technique identifies the code smell type by using adaboost classifier in source code where the rectification is needed through constructing a decision tree. Initially, each software code considered as input and detects the code smell. After that, the number of code smell types is identified. The refactoring technique is used for rectifying the code smell in source code.

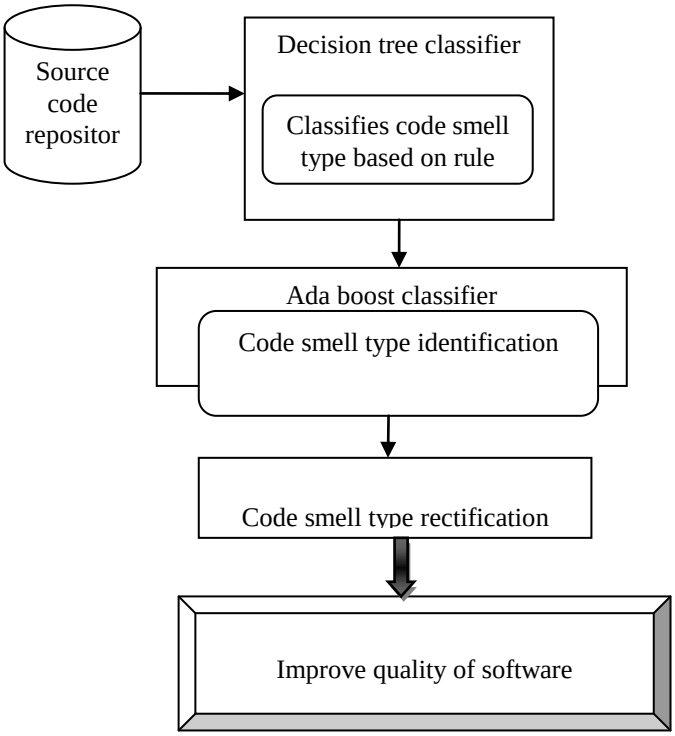

Figure 1 Processing diagram of Machine Learning Ada-Boost Classifier technique

As a result, the software quality is increased with minimum cost. The brief description of ML-ABC technique is explained in the following sections.

A. Machine Learning Ada-Boost Classifier For Code Smell Type Identification

AdaBoost is a boosting classifier whose basic process is to select and combine a group of weak classifiers to form a strong classifier. Adaboost classifier includes a group of weak (i.e. base) classifiers means that the classification results of individual classifier is comparatively reduced. In order to improve the classification, Adaboost classifier combines a several "weak classifiers" into a single "strong classifier" to classify the code smell type in software code. ML-ABC technique uses the decision tree as the weak classifier to classify the code smell types effectively. The process involved in AdaBoost with decision tree classifier is shown in figure 2.

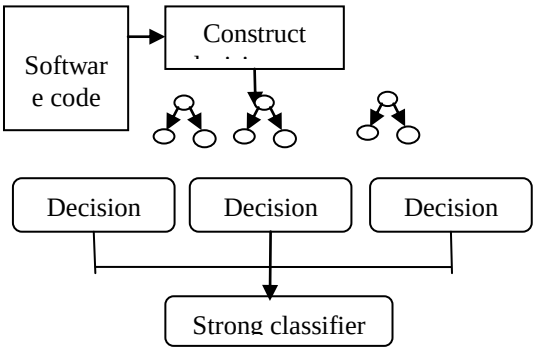

Figure 2 Flow processing diagram of AdaBoost with decision tree classifier

Figure 2 shows the AdaBoost with decision tree classifier to improve identification accuracy with minimum cost. The input to classifier is a training data (i.e. source code). Generally, there are many types of software code smells are presented. This code smell type is identified through classification based on decision. A typical decision tree is constructed with a root node and leaf nodes. Each node has decision to perform efficient classification. The root node considers the source code line which performs the classification based on the rule. Therefore, the root node takes the decision to classify the code smell type. The leaf node is also called as terminal node of the tree with a class label. From the decision tree, each path from the root node to leaf node constructs a classification rule. The numbers of rules in the classification for identifying the types of code smells such as too many parameters, runtime class type detection, class having little functions, class having more functions, Redundant code, code never process at running time, repeated code, unused code and so on. Based on this rule, the type of code smell is classified. If source code has the rule of "several parameters", then code smell type is classified as Long method. If source code contains rule of "Redundant code", then the classifier is classified as duplicate code smell type. Similarly, all the types of code smell are classified for rectification.

Let us consider $\{(x_1,Y_1),(x_2,Y_2),\dots (x_n,Y_i)\}$ is a set of input data i.e. source code and 'Y' is a target output class (i.e. code smell type). The ada-boost classifier function provides the class output $Y_i = \{-1,+1\}$, , where $Y_i = -1$ is incorrectly classified code smell type and $Y_i = +1$ represents code smell are correctly classified. Therefore, a decision tree is a classifier which performs recursive classification to identify all types of software code smell. The Ada-boost classifier is applied to construct a strong classifier. By applying Ada-boost classifier, the weight of the all the source code is initialized as follows,

$$\sum_{i=1}^{n} w_i(t) = \frac{1}{n} \quad \text{_____(1)}$$

From (1), $w_i(t)$ denotes a weight function of source code and 'n' denotes size of source code. The weights values are assigned based on classification error of base decision tree classifier. In order to find weak learner $h_t(x_n)$ that minimizes error, the weighted sum error in base classifier is measured as follows,

$$e_R = \sum_{i=1}^{n} w_i(t) \quad \text{_____ (2)}$$

From (2), where e_R denotes a error rate which is measured as follows,

$$\omega_t = 0.5 \ln\left(\frac{1-e_R}{e_R}\right) \quad \text{_____(3)}$$

From (3), ω_t represents the adjustment coefficient to obtain the final classification result. Subsequently, base classifier weight is updated as follows,

$$w_i(t+1) = \frac{w_i(t) e^{(-\omega_t Y_i h_t(x_n))}}{\alpha} \quad \text{_____ (4)}$$

From (4), $w_i(t+1)$ represents updated weight and $w_i(t)$ denotes an original weight. Here, $h_t(x_n)$ denotes the prediction class labels of the n^{th} base decision tree classifier and $'Y_i'$ is strong classifier output result, α represents the normalization factor. After that, the updated weight of decision tree classifier is normalized as,

$$\sum_{i=1}^{M} w_i\ (t+1) = 1 \quad \text{_____ (5)}$$

From (5), the weight of weak decision tree classifier is compared with the certain threshold value (δ). If weight of weak decision tree classifier is higher than the threshold value, then the output results of all weak decision tree classifier is combined into a strong classifier using Ada booster technique for improving the code smell type identification accuracy. The strong classifier output is expressed as,

$$Y_i = sign\ (\sum_{t=1}^{T} \omega_t\ h_t(x_n)) \quad \text{_____ (6)}$$

From (6), $'Y_i'$ represents the output of strong classifier which provides the two results to enhance the accuracy of classifier. From the equation (6), represents positive and negative results of the output classifier. The output of strong classifier is described as follows,

$$Y_i = \begin{cases} +1 & correctly\ classified \\ -1 & incorrectly\ classified \end{cases} \quad \text{_____ (7)}$$

From (7),Yi=+1 where denotes the code smell type is correctly classified and denotes a code smell type is incorrectly classified. Based on the classification results, software code smell type is identified. The algorithmic process of AdaBoost with decision tree classifier is described as follows.

Input: $\{(X_1,Y_1),(X_2,Y_2),\dots (X_n,Y_n)\}$ is a set of training sample
Output: Code smell type identification accuracy
Step 1: Begin
Step 2: For each source code
Step 3: Classify the code smell type based on rule using decision tree
Step 4: Initialize the weight of source code using (1)
Step 5: Calculate weighted sum error in base classifier using (2) (3)
Step 6: Update the base classifier weight using (4)
Step 7: if $(w_t > \delta)$ then
Step 8: Train a base decision tree classifier for classifying the code smell type
Step 9: if strong classifier output result $Y_i = +1$ then
Step 10: Code smell type is correctly classified
Step 11: else
Step 12: Code smell type is incorrectly classified
Step 13: End if
Step 14: End if
Step 15: End for
Step 16: End

Figure 3 AdaBoost with decision tree classifier

Figure 3 shows the AdaBoost with decision tree classifier algorithm for classifying the code smell type effectively. This helps to identify which type of software code smell is presented in source code. Initially, the base decision tree classifier is applied to construct the tree based on decision. If the source code has the specific rule, then the classifier is classified as a code smell type. In order to improve the classification performance, AdaBoost classifier is used to strong the base classifier.

For each source code, weight function of base classifier is measured.

If the weight value is higher than the specific threshold value then the base decision tree classifier makes as strong classifier through Ada boost classifier. As a result, the strong output classifier provides as result +1 which indicates the code smell type is correctly classified. If the strong output classifier provides as result -1 means the classifier is incorrectly classified as code smell type. As a result, the code smell type identification accuracy is increased with less false positive rate.

B. Code Smell Type Rectification

After identifying the code smell type in a source code, rectification is performed through the refactoring techniques. Refactoring is a technique used to remove the particular types of code smell in source code. Software Refactoring is also used to reduce the cost of rectification through changing the internal behavior of the source code. It also improves the software code quality and maintainability. Source code design and it quality are enhanced using refactoring and also increases the code reusability. Therefore, the refactoring creates a source code easier to understand and it helps to run a program fast.

Figure 4 shows the code type rectification using refactoring technique. The software developer identifies the code smell type using machine learning Adaboost classifier. The identified code smell type is

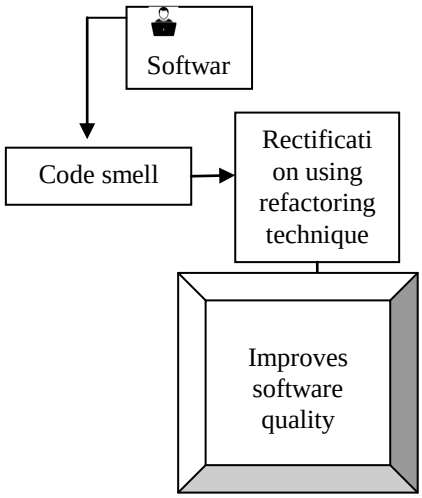

Figure 4 Software code smell type rectification

corrected by using exact refactoring method. Generally, there are several refactoring methods are available such as Extract Method, Inline Method, Replace Temp With Query, Replace Method With Method Object, Move method, Extract Class, Replace Type Code with Class, Remove parameter, Add parameter and so on.

The above AdaBoost classifier identifies the different code smell types such as long method and redundant code. Long Method code smell has long parameter and it's too lengthy, hence it is complex to read the software code. Then the suitable refactoring methods such as decompose conditional, Extract Method, replace method with method object, and replace Temp with Query are used to remove this kind of code smell and replace it. In addition, other type of code smell is a redundant code. The source code contains the redundant code smell which is the worst smell. The developer identifies this duplication code in source code in a simple manner. The proposed ML-ABC technique uses the refactoring method as Extract Class, Extract Method, Form Template Method, and Pull Up Method to rectify the duplication type of code and remove unused code and repeated code in source code. This helps to reduce the space complexity. As a result, the refactoring method is used for the entire source code where the code smell type is identified. This helps to improve the software program quality with minimum cost.

III. EXPERIMENTAL EVALUATION

The proposed MLABC technique is experimented using JAVA programming code with SchoolMate dataset which contains the open-source programs. These open-source programs are considered as source code and the types of code smells are identified and rectified through refactoring. The SchoolMate dataset is taken from [21]. The SchoolMate consists of brief solution related to elementary, middle and high schools.

It processed with four domains, namely, administration, teachers, students and parents, where the administration manages both the classes and users of the SchoolMate whereas the teachers' manages the details about assignments and grades.

The MLABC technique zexperimented against monitor-based instant refactoring framework [1] and feedback-based approach [2]. The experiment is conducted on the factors such as code smell type Identification accuracy, false positive rate, code smell type rectification cost and space complexity.

IV. RESULTS AND DISCUSSION

Analysis of MLABC technique is performed and compared with existing monitor-based instant refactoring framework [1] and feedback-based approach [2]. The analysis is carried out on the factors such as Code smell type Identification accuracy, false positive rate, Code smell type rectification cost and space complexity. The performance is evaluated according to the following parameters with the help of table and graph values.

A. Impact of Code smell type Identification accuracy:

Code smell type Identification accuracy is defined as the number of code smell types in source code lines are correctly identified through classification to the number of source code lines. It is measured in terms of percentage (%). The mathematical formula for classification accuracy is defined as follows,

$$CMTIA = \frac{N - No.of\ code\ smell\ type\ is\ correctly\ identified\ \ in\ source\ code\ lines}{N} * 100$$

---(8)

From (8), Where $CMTIA$ denotes a Code smell type identification accuracy and 'N' denotes a number of source code lines.

Table 1 Tabulation for Code smell type

Source code (KB)	Code smell type Identification accuracy (%)		
	MLABC	Monitor-based instant refactoring framework	Feedback-based approach
2	85	62	71
4	86	65	73
6	87	69	76
8	88	70	78
10	90	72	82
12	92	75	83
14	93	78	84
16	94	80	85
18	95	82	86
20	96	84	87

Table 1 describes a Code smell type Identification accuracy with respect to size of source code (KB). The input source code contains multiple lines. From the source code lines, numbers of code lines where the code smell are detected for rectification. Code smell type Identification accuracy is increased using MLABC technique when compared to existing monitor-based instant refactoring framework [1] and feedback-based approach [2].

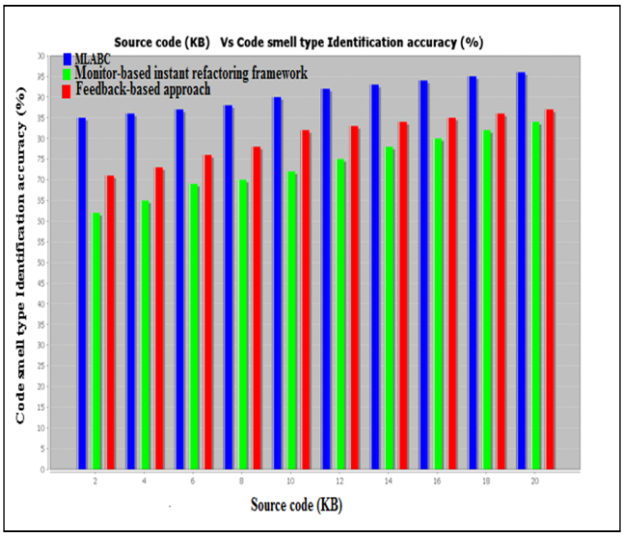

Figure 5 Measure of code smell type identification

accuracy tree classifier is used certain rule for making the decision. It classifies the code smell types according to the specific rule. For example, if the source code contains the redundant code, then the decision tree classifier classified the code smell as duplicate code smell. After that, ada-boost classifier is used to make a base classifier into a strong based on the weight value. If the weight value of base classifier is greater than threshold value, then the weak classifier is combined to create a strong classifier. The strong classifier output results used to identify the types of code smells in source code lines. As a result, the code smell type identification accuracy is considerably increased by 24% and 13% compared to existing monitor-based instant refactoring framework [1] and feedback-based approach [2] respectively.

B. Impact of false positive rate

False positive rate is defined as the ratio of number of code smell type in source code lines are incorrectly identified by applying classification to the number of source code lines. It is measured in terms of percentage (%). The formula for false positive rate is expressed as follows,

$$FPR = \frac{No.of\ code\ smell\ type\ is\ incorrectly\ identified\ in\ source\ code\ lines}{N} * 100 \quad \text{---- (9)}$$

Table 2 Tabulation for False positive rate

Source code (KB)	False positive rate (%)		
	MLABC	Monitor-based instant refactoring framework	Feedback-based approach
2	**22**	**42**	**31**
4	**25**	**44**	**33**
6	**27**	**45**	**34**
8	**28**	**47**	**35**
10	**30**	**50**	**38**
12	**32**	**52**	**40**
14	**34**	**53**	**42**
16	**36**	**55**	**44**
18	**38**	**57**	**46**
20	**40**	**58**	**48**

Table 2 describes a results analysis of false positive rate with three different methods MLABC technique and existing monitor-based instant refactoring framework [1] and feedback-based approach [2]. Let us consider the size of source code is 2KB, the source code lines where the code smell type is incorrectly identified. The performance of false positive rate is significantly reduced by applying MLABC technique when compared to existing monitor-based instant refactoring framework [1] and feedback-based approach [2].

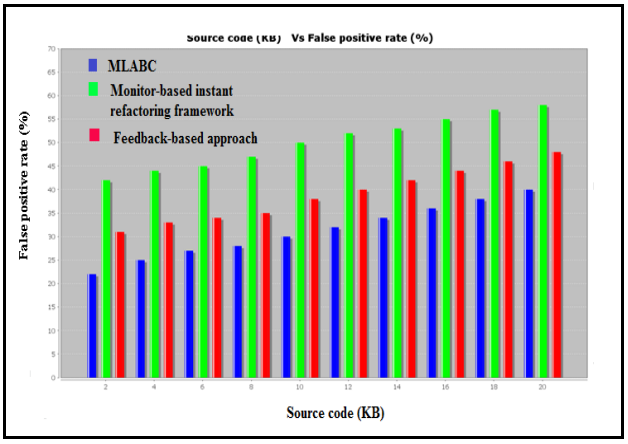

Figure 6 Measure of false positive rate

Figure 6 depicts the result analysis of the false positive rate with respect to source code. The figure clearly shows that the proposed MLABC technique classified the code smell type with less false positive rate compared to existing methods. This is because, the MLABC technique effectively classified the code smell type based on decision rule. For each iteration, the ada-boost classifier minimizes the total weighted error of the base classifier. The MLABC improves the classification which resulting in higher identification accuracy.

If output of strong classifier provides the negative results, then the code smell types is incorrectly identified. But the proposed MLABC technique perform efficient decision rule to classify the code smell type with less false positive rate. Let us consider the input of 2KB source code, the false positive rate of proposed method is 22% whereas existing monitor-based instant refactoring framework and feedback-based approach are 42% and 31% respectively. This shows the better improvement of MLABC technique. As a result, the false positive rate is considerably reduced by 38% and 21% using MLABC technique when compared to existing monitor-based instant refactoring framework [1] and feedback-based approach [2] respectively.

C. Impact of Code smell type Rectification cost

Code smell type rectification cost is measured based on amount of time required to rectify the code smell type in source code program by using refactoring meth The formula for rectification cost is measured as

$$CSTRC = n * time\ (rectify\ the\ code\ smell\ type\ ) \quad \text{---} \quad (10)$$

From (10), where $CSTRC$ denotes Code smell type rectification cost, $'n'$ represents number of code smell types in a lines. It is measured in terms of millisecond (ms).

Table 3 Tabulation for Rectification cost

Source code (KB)	Code smell type Rectification cost (ms)		
	MLABC	Monitor-based instant refactoring framework	Feedback-based approach
2	9	20	15
4	12	30	22
6	18	42	34
8	25	54	45
10	33	63	52
12	42	79	67
14	58	87	74
16	69	107	82
18	77	117	95
20	84	128	103

Table 3 describes the experimental results of Code smell type rectification cost using three different methods MLABC technique and existing monitor-based instant refactoring framework [1] and feedback-based approach [2]. By using MLABC technique, the number of code smell in the source code is correctly identified and rectified with minimum cost. Therefore, MLABC technique reduces the code smell rectification cost compared to existing methods.

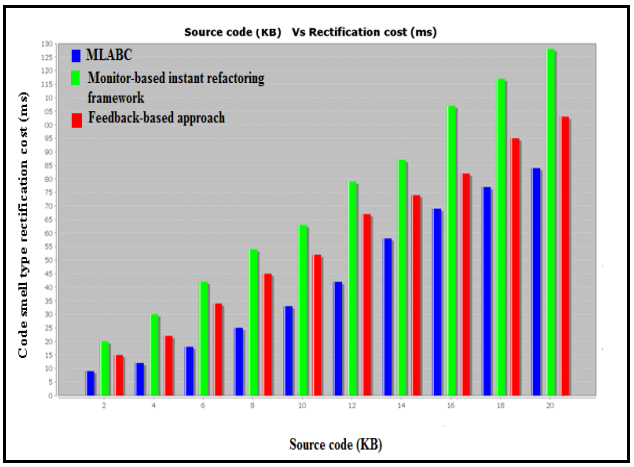

Figure 7 Measure of Code smell type rectification cost

Figure 7 illustrates the performance results of Code smell type rectification cost in terms of time for rectifying the code smell in source code with respect to source code size from 2KB to 20KB. The code smell type rectification cost is reduced compared to existing methods as shown in figure 6. Due to, the MLABC technique uses the suitable refactoring technique for rectifying the identified software code smell in source code lines. Initially, adaboost with decision tree classifier classifies the code smell type. This helps to identify the software code smell in source code. After that, the software code smell is rectified by using suitable refactoring method. This process is repeated for the entire software code with minimum cost. Let us consider, the 2KB input code of software program, the cost of MLABC technique is 9ms whereas 20ms and 15ms cost for existing monitor-based instant refactoring framework [1] and feedback-based approach [2]. Therefore, the MLABC technique improves the software quality by identifying and rectifying the code smell type with minimum cost. As a result, code smell type rectification cost is reduced by 46% and 33% when compared to existing monitor-based instant refactoring framework [1] and feedback-based approach [2] respectively.

D. Impact of space complexity

Space complexity is defined as an amount of space consumed by the source code after rectification (i.e. removing the repeated code and replacing the code with better code). The space complexity is measured as follows,

$$SC = space\ consumed\ (after\ rectification) \quad \textbf{(11)}$$

From (11), Where 'SC' denotes a space complexity and it is measured in terms of bytes. Lower the space complexity, more efficient the method is said to be. Table 4 shows an experimental result of space complexity with respect to software code. The space complexity is the amount of storage space consumed of an algorithm after the code smell type rectification. From the table value, the space complexity using MLABC technique is considerably reduced when compared to existing monitor-based instant refactoring framework [1] and feedback-based approach [2] respectively.

Table 4 Tabulation for Space complexity

Source code (KB)	Space complexity (Bytes)		
	MLABC	Monitor-based instant refactoring framework	Feedback-based approach
2	1636	1869	1812
4	3356	3942	3897
6	5036	5978	5879
8	7124	7852	7542
10	8365	9750	9123
12	10987	11912	11754
14	13220	13812	13698
16	14231	15913	15852
18	16895	17324	17130
20	18525	19356	19120

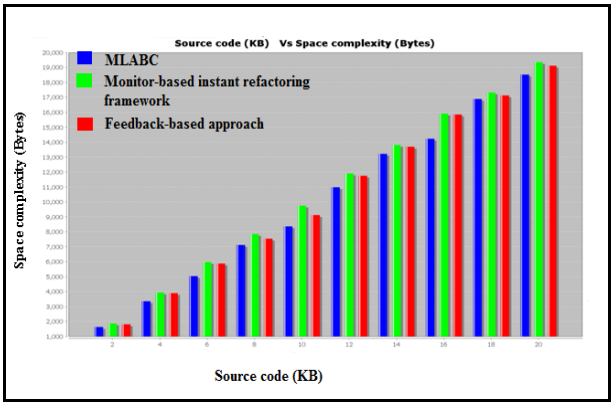

Figure 8 Measure of space complexity

Figure 8 illustrates the performance results of space complexity versus source code. The proposed MLABC technique consumed less amount of storage space after code smell rectification when compared to existing methods. This is because, MLABC technique effectively identifies the code smell in source code program by performing the classification process with the help of adaboost classifier. This helps to efficiently identify the code smell in the source code program for rectification. Identification of those code smells are removed and replaced the unused code from source code by performing the rectification process, hence the space complexity is also reduced. As a result, MLABC technique uses less amount of memory space for storing the multiple cloud user data on cloud server. Therefore, the space complexity is considerably reduced by 10% and 8% when compared to existing monitor-based instant refactoring framework [1] and feedback-based approach [2] respectively

As a result, the performance results of MLABC technique improves the code smell type identification accuracy and reduces the false positive rate with minimum rectification cost and space complexity.

V. RELATED WORKS

A flexible and lightweight approach based on multiple searching techniques was developed in [11] for the detection of code smells from source code of multiple languages. However, it failed to focus on detecting other types of code smells. Hence, the MLABC technique effectively identifies the code smell type using ada boost classifier. A two-step automated technique was introduced in [12] to detect and resolve the different kinds of maintainability faults in source code. However, more badly-designed code in source code was not detected. Hence, MLABC technique detects number of code smell type in source code. Investigation of static and dynamic software metrics with different rules were performed in [13] to identify a bad smells. Though the code smell was identified, a suitable refactoring was not performed. Hence, the MLABC technique uses suitable refactoring method to improve software quality with minimum cost. The window base GUI application was introduced in [14] to detect code smell. But, refactoring methods was not used on the basis of calculated metrics on source code. Therefore, MLABC technique effectively rectifies the code smell in source code using refactoring technique. In [15], a different code smells were identified through the application of various machine learning algorithms. But, code smell type classification was not performed. Though the method detect the code smell, a software quality assessment tasks was not performed. Hence, MLABC technique increases the software quality by using refactoring methods. A metric based approach was developed in [16] for software code clone detection. However, the code smell detection accuracy was not improved at required level. MLABC technique achieves high code smell detection accuracy with minimum cost. A hybrid clone detection scheme was introduced in [17] that design and investigate a hybrid technique for identifying the software clone in an application. But, it has high false positive rate. Hence, MLABC technique reduces the performance of false positive rate through classification. A clone detection algorithm was presented in [18] to detect duplicate code for UML domain. However, it failed to improve the accuracy of clone detection. Therefore, a MLABC technique significantly increases the code smell type identification accuracy. The program dependence graph (PDG) was developed in [19] for identifying the software clones. But it was failed for detecting exact bugs in program. The MLABC technique efficiently identifies the code smell type in source code program by using AdaBoost classifier. A new refactoring methods and metrics were described in [20] to recognize the characteristics of bad smells in source code. However, the cost minimization during the refactoring remained unsolved. MLABC technique reduces the performance of rectification cost.

VI. CONCLUSION

An efficient Machine Learning Ada-Boost Classifier (MLABC) technique is developed for software code smell type identification and rectification with cost minimization. Adaboost with decision tree classifier is used to improve the software quality. MLABC technique constructs a base decision tree classifier classifies the code smell type by using certain rule. If the source code has a rule which related to code smell, then the classifier classifies the types of code smell is presented. In order to improve the classification, Ada-Boost machine learning technique is used. After that, the identified code smells are rectified by applying the refactoring technique with minimum cost and space complexity.

Experimental evaluation is conducted using schoolmate dataset to evaluate the proposed MLABC technique in terms of code smell type Identification accuracy, false positive rate, and code smell type rectification cost and space complexity. The results analysis of MLABC technique improves software code smell type identification accuracy with minimum false positive rate, code smell type rectification cost and space complexity than the state-of-art methods.

ACKNOWLEDGEMENT

We would like to thanks Department of Computer Science(No.F.5-6/2018/SAP-II) Periyar University for providing me the Opportunity to study as a research scholar. We are also grateful to the colleague and family members for their continuous support.

REFERENCES

1. Hui Liu , Xue Guo, Weizhong Shao, "Monitor-Based Instant Software Refactoring", IEEE Transactions on Software Engineering, Volume 39, Issue 8, 2013, Pages 1112 – 1126
2. Hui Liu, Qiurong Liu, Zhendong Niu , Yang Liu, "Dynamic and Automatic Feedback-Based Threshold Adaptation for Code Smell Detection", IEEE Transactions on Software Engineering Volume 42, Issue 6, 2016, Pages 544 - 558
3. Ah-Rim Han and Doo-Hwan Bae , "Dynamic profiling-based approach to identifying cost-effective refactorings", Information and Software Technology, Elsevier, Volume 55, 2013, Pages 966–985
4. Harris Kristanto Husien, Muhammad Firdaus Harun, Horst Lichter, "Towards a Severity and Activity based Assessment of Code Smells", Procedia Computer Science, Elsevier, Volume 116, 2017, Pages 460–467
5. Alexander Chatzigeorgiou, Anastasios Manakos, "Investigating the evolution of code smells in object-oriented systems", Innovations in Systems and Software Engineering, Springer, Volume 10, Issue 1, 2014, Pages 3–18
6. Santiago A. Vidal , Claudia Marcos, J. Andrés Díaz-Pace, "An approach to prioritize code smells for refactoring", Automated Software Engineering, Springer, Volume 23, Issue 3, 2016, Pages 501–532
7. Dag I.K. Sjøberg , Aiko Yamashita , Bente C.D. Anda , Audris Mockus , Tore Dybå, "Quantifying the Effect of Code Smells on Maintenance Effort", IEEE Transactions on Software Engineering, Volume 39, Issue 8, 2013, Pages 1144 – 1156
8. Sandeep Kaur and Harpreet Kaur, "Identification and Refactoring of Bad Smells to Improve Code Quality", International Journal of Scientific Engineering and Research (IJSER), Volume 3 Issue 8, August 2015, Pages 99-102
9. Thanis Paiva , Amanda Damasceno, Eduardo Figueiredo and Cláudio Sant'Anna, "On the evaluation of code smells and detection tools", Journal of Software Engineering Research and Development, Springer, Volume 5, Issue 7, Pages 1-28
10. Wael Kessentini , Marouane Kessentini , Houari Sahraoui , Slim Bechikh , Ali Ouni, "A Cooperative Parallel Search-Based Software Engineering Approach for Code-Smells Detection", IEEE Transactions on Software Engineering , Volume 40, Issue 9, 2014, Pages 841 – 861
11. Ghulam Rasool, Zeeshan Arshad, "A Lightweight Approach for Detection of Code Smells", Arabian Journal for Science and Engineering, Springer, Volume 42, Issue 2, 2017, Pages 483–506
12. Ali Ouni, Marouane Kessentini, Houari Sahraoui, Mounir Boukadoum, "Maintainability defects detection and correction: a multi-objective approach", Automated Software Engineering, Springer, Volume 20, Issue 1, 2013, Pages 47–79
13. Sukhdeep Kaur and Raman Maini, "Analysis of Various Software Metrics Used To Detect Bad Smells", The International Journal Of Engineering And Science (IJES), Volume 5 , Issue 6, 2016, Pages 15-19
14. Anshu Rani, Harpreet Kaur, "Detection of bad smells in source code according to their object oriented metrics", International Journal for Technological Research in Engineering, Volume 1, Issue 10, 2014, Pages 1211-1214
15. Francesca Arcelli Fontana , Mika V. Mäntylä , Marco Zanoni , Alessandro Marino, "Comparing and experimenting machine learning techniques for code smell detection", Empirical Software Engineering, Springer, Volume 21, Issue 3, 2016, Pages 1143–1191
16. Sushma and Jai Bhagwan, "A Novel Metrics Based Technique for Code Clone Detection", International Journal Of Engineering And Computer Science, Volume 05, Issue 9, 2016, Pages 18221-18224
17. Priyanka Batta and Himanshi, "Hybrid technique for software code clone detection", International Journal of Computers & Technology, Volume 2, Issue 2, 2012, Pages 98-102
18. Harald Störrle, "Towards clone detection in UML domain models", Software & Systems Modeling, Springer, Volume 12, Issue 2, 2013, Pages 307–329
19. S. Sargsyan, Sh. Kurmangaleev, A. Belevantsev, and A. Avetisyan, "Scalable and Accurate Detection of Code Clones", Programming and Computer Software, Volume 42, Issue 1, 2016, Pages 27–33.
20. Karnam Sreenu and Jagannadha Rao, "Performance - Detection of Bad Smells In Code for Refactoring Methods", International Journal of Modern Engineering Research (IJMER), Volume 2, Issue 5, 2012, Pages 3727-3729
21. Schoolmate dataset: https://sourceforge.net/projects/schoolmate/?source=directory

AUTHORS PROFILE

M. Sangeetha, received her B.Sc & M.Sc. degree in Computer Science from Bharathidasan University, Trichy in 1994 and 1999 respectively, also received her M.Phil degree in Periyar University in 2007. Currently she is pursuing her full time research in Computer Science at Periyar University Salem-11. From 1999 to 2014, she was the Faculty in the Department of computer science, Salem Sowdeswari College, Salem. From 2014 to 2015, she acted as a Vice Principal in Sakthikailash Women's College, Dharmapuri. Her Professional activities include guided Seven M.Phil Scholars in the field of CS. Also she has Published and presented more than 15 Papers in International and National Journals and also in Conferences. Her current research interests on Software Engineering. She is a Life Member of the IAENG, Hong Kong.

Dr. C. Chandrasekar, is the Professor of the Department of Computer Science at Periyar University. He pursued his M.C.A., and awarded Ph.D. His Research Areas are Mobile and Wireless Computing/ 10/521.He have 25 years of teaching experience and 15 years of research interest And he published book on Chandrasekar. C and Venkatesan. K., '' Introduction to Information Technology",Serve the People Publications, 2010.