Received February 26, 2019, accepted March 8, 2019, date of publication March 14, 2019, date of current version April 3, 2019.

*Digital Object Identifier 10.1109/ACCESS.2019.2905133*

# Assessment of Code Smell for Predicting Class Change Proneness Using Machine Learning

**NAKUL PRITAM[1], MANJU KHARI[2], LE HOANG SON[3,4], RAGHVENDRA KUMAR[5], SUDAN JHA[6], ISHAANI PRIYADARSHINI[7], MOHAMED ABDEL-BASSET[8], AND HOANG VIET LONG[9,10]**

[1]Leading Pseudo Code Labs, Delhi 110012, India
[2]Department of Computer Science and Engineering, AIACTR, New Delhi 110031, India
[3]VNU Information Technology Institute, Vietnam National University, Hanoi 010000, Vietnam
[4]College of Electronics and Information Engineering, Sejong University, Seoul 100083, South Korea
[5]Department of Computer Science and Engineering, LNCT College, Jabalpur 482053, India
[6]School of Computer Engineering, KIIT University, Bhubaneswar 751024, India
[7]Department of Electrical and Computer Engineering, University of Delaware, Newark, DE 19716, USA
[8]Department of Operations Research and Decision Support, Faculty of Computers and Informatics, Zagazig University, Zagazig 44159, Egypt
[9]Division of Computational Mathematics and Engineering, Institute for Computational Science, Ton Duc Thang University, Ho Chi Minh City 700000, Vietnam
[10]Faculty of Mathematics and Statistics, Ton Duc Thang University, Ho Chi Minh City 700000, Vietnam

Corresponding author: Hoang Viet Long (hoangvietlong@tdtu.edu.vn)

**ABSTRACT** Assessment of code smell for predicting software change proneness is essential to ensure its significance in the area of software quality. While multiple studies have been conducted in this regard, the number of systems studied and the methods used in this paper are quite different, thus, causing confusion for understanding the best methodology. The objective of this paper is to approve the effect of code smell on the change inclination of a specific class in a product framework. This is the novelty and surplus of this work against the others. Furthermore, this paper aims to validate code smell for predicting class change proneness to find an error in the prediction of change proneness using code smell. Six typical machine learning algorithms (Naive Bayes Classifier, Multilayer Perceptron, LogitBoost, Bagging, Random Forest, and Decision Tree) have been used to predict change proneness using code smell from a set of 8200 Java classes spanning 14 software systems. The experimental results suggest that code smell is indeed a powerful predictor of class change proneness with multilayer perceptron being the most effective technique. The sensitivity and specificity values for all the models are well over 70% with a few exceptions.

**INDEX TERMS** Code smell, change proneness, software maintenance, machine learning, multilayer perceptron.

## I. INTRODUCTION

**Change in software** is one of the most unpredictable situations which may come across in the lifespan of the system [3]. It is much tied to software design and theories suggesting best practices rather than specifying exactly how a design must be made [2]. Many researches have been conducted in the past to quantify attributes of a software system using patterns and metrics in order to evaluate good and bad aspects of the software and to predict possible changes [5], [14]. Recent studies have indicated that **code smell can predict the change proneness** more accurately than the static code metrics [6]. *Code smell*is a bad implementation choice in the design phase and it becomes obvious in the implementation phase [16]. Good implementation choices are called design patterns while bad ones are called anti-patterns [15].

Fowler [1] defined 22 code smells and suggested areas where refactoring may be applied. It is impossible to examine a class for all code smell without using a threshold. For example, the smell Child Class cannot be determined without using a threshold such as NOM, LOC and Number of Variables [10]. Kaur and Jain [7] reviewed six machine learning algorithms with two modes (random-under sampling before feature selection, and feature selection before random-under sampling)for suspecting changes incline utilizing the code smell. Results indicated that random under sampling before feature selection is the most effective strategy with Gene Expression Programming while cascade correlation network and Tree-Boost are among the top algorithms in predicting change proneness [7].

The associate editor coordinating the review of this manuscript and approving it for publication was Victor Hugo Albuquerque.

Vidal *et al.* [16] developed a tool for ranking of code smell based on past segment alterations, vital modifiability situations for the framework, and significance of the sort of smell. They are complementary to assess the smell from different perspectives. Saboury *et al.* [15] detected 12 types of code smell in 537 releases of five popular JavaScript applications with the aim to understand how they impact the fault-proneness of applications. Hecht *et al.* [4] conducted an exact examination concentrating on the individual and joined execution effects of three Android performance code smell (namely, Internal Getter/Setter, Member Ignoring Method, and HashMap Usage) on two open source Android apps. Ma *et al.* [9] researched the likelihood of enhancing the execution of fault detection by utilizing code smell detection and Cohen's Kappa statistics. Hadj-Kacem and Bouassida [17] proposed a hybrid approach to detect code smell using deep learning Auto-encoder and Artificial Neural Network algorithms. The evaluation was performed using parameters precision, recall and F-measure. However, there were several issues regarding validation of the experiment, such as Internal validity, conclusion validity and external validity. Fontana *et al.* [18] used a machine learning approach for code smell detection. The research paper focused on six classifiers evaluated on basis of accuracy, ROC curves and F-measure. Azadi *et al.* [19], suggested machine learning based code smell detection through tool WekaNose. Although the research paper offered a novel approach to select algorithms for classification of an instance using an experimental approach, the article lacked several technicalities and experimental analysis. Nucci *et al.* [20] performed yet another research on detecting code smell using machine learning techniques. The performances have been evaluated on basis of accuracy, F-measure and AUC-ROC. However, the research suffers from several threats to construct validity, external validity and conclusion validity. Other researches on code smell can be retrieved in [11], [12], [13], [21], and [22].

In this paper, we **aim to validate Code Smell for Predicting Class Change Proneness** to find error in prediction of change proneness using code smell. For understanding the overall **idea**, in the first step two versions of 14 software's have been compared to find out the classes that are change prone. For making the datasets of these 14 open-source projects, initially, it was considered to divide the software data into proper class relations and considerable amount of metrics of the software. The dataset contains different softwares that were used in previous researches where other algorithms were used to predict changes. We also check if different versions of software are present. Later, ten-cross validation has been used to divide the dataset into ten subsets among which nine for training machine learning model. Six typical machine learning algorithms (Naive Bayes Classifier, Multilayer Perceptron, LogitBoost, Bagging, Random Forest, and Decision Tree) have been used to predict change proneness using code smell. The predicted values are then compared and analyzed against the actual Change Prone classes using ROC analysis to get the false positives and true positives. We observe that most of the research conducted in the past does not rely on huge datasets. While some articles lack experimental analysis, others apprise that change in parameters would alter the results. Further, the evaluation metrics are limited for many of the code smell researches conducted in the past. This paper evaluates the code smell not only on basis of ROC curves but also provide values for sensitivity, specificity, cut-off point and Area under Curve for better evaluation.

The **difference and novelty** of this research against the previous studies are shown below. The previous works such as the studies by Khomh *et al.* [8], Gatrell and Counsell [3] have shown significant relations between smells and code change. However, their empirical studies did not target for large datasets and because of that some issues like impact of code smells i.e. detected vs. manually validated; limited sizes; lack of empirical analysis on observed specific smell type; the optimal value for the effects on defected code smell etc. have not been solved completely. In order to handle this, going through huge datasets is much needed because until and unless different kinds of datasets are used, applying machine learning may be futile. Besides, there was no empirical evaluation between code smells and code fault proneness in the previous works. Also, most of the works have been explored based on only one or two software projects. This work is based on consideration of large support systems like SAP and ERP because these huge datasets guarantee smell detectors to be effectively achieved with high accuracy. Unlike the previous papers, this paper also gives the intrinsic precision affects in a very high probability. Most importantly, we have relied on data that may be experimentally conducted on various open source systems.

The **advantages** of this research can be broadly listed as below. Considering the 5 major advantages below, this work also targets analyzing percentage of code components in a system and effect on change- and fault-proneness on a large set of software projects.

1. Empirical relationship between the code smells in huge scale software.
2. Notable correlation between smells & proneness.
3. Since unlimited size of open source based data sets are used, the impact of code smells on change- and fault-proneness is highly significant.
4. More in depth analysis for harmful smells.
5. High maintainability.

The rest of this paper is organized in the following manner. Section 2 provides the experimental datasets and the machine learning methods. Section 3 presents the experimental results and discussion. Section 4 highlights conclusions and future works.

## II. RESEARCH QUESTIONS

There may arise particular quality issues because of code decay if changes are caused by defect repair. Identifying where most changes are set aside, in a few minutes the system may recognize key change-prone classes and key change-prone interactions. Changes may originate from a single defect report. Likewise, accumulations of classes that experience changes together may be made obvious, since they are prime focuses for ventured up perception and conceivably change endeavors. Moreover, this investigation includes a number of change classes. It is vital to analyze whether change prone clusters of classes are in relation to design structures and change-related interactions. While changes in singular classes can be tallied, this does not uncover critical parts of code changes. Knowing which classes can be extremely useful, change-proneness may demonstrate particular hidden quality issues. The research questions we examined are:

1. Is there an approach to recognize the most change-inclined accumulations of classes in a framework?
2. Can change-inclination recognize nearby change-inclinations due to the change in collaborations of classes?
3. Do singular changes made in light of one change ask for influence of the classes that are connected in the intelligent outline of a framework? Are there verifiable associations between framework components that are not some portion of any plan portrayal?
4. How would we make the data that came from the change-inclination noticeable?

## III. MATERIALS AND METHODS

### A. DATASETS

The entire process can be divided into three parts:

1) Data acquisition and processing;
2) Change and smell estimation;
3) Analysis using Machine Learning Methods.

#### 1) DATA ACQUISITION AND PROCESSING

The datasets for this study were collected as follows. Firstly, we downloaded two stable releases of each system as listed in Table 1. Pre-processing of datasets included removing all Java files in the versions that were not compatible with the current one. All 14 systems combined had approximately 10000 classes in two versions of each system out of which more than 8200 were left after pre-processing. This essentially means that we are left with around 4100 unique classes spanning over 14 software systems.

#### 2) CHANGE AND SMELL ESTIMATION

In this step, the actual change of a class undergoes in two versions is calculated to obtain the metric values for each class in each system. They are then used to analyze each class for code smell. To calculate the exact change of a class that has gone through in the two stated versions, we used an open-source tool named CLOC to examine two versions of the same file and give as outputs below:

a) Number of Lines that are unchanged.
b) Number of Lines added to the prior version.
c) Number of Lines deleted from the prior version.
d) Number of Lines modified over the two versions.

These outputs are then used to calculate the *amount of change* as:

**TABLE 1.** Summary of the Datasets.

| No. | Name | Ver. 1 | Ver. 2 | P/L Used | Total LOC | Total Classes | Classes Exhibiting Change | Classes Without Change |
|---|---|---|---|---|---|---|---|---|
| 1 | AOI | 2.0 | 2.9 | Java | 58260 | 249 | 202 | 47 |
| 2 | CheckStyle | 5.2 | 5.5 | Java | 50461 | 693 | 145 | 548 |
| 3 | FreePlane | 1.1.1 | 1.1.3 | Java | 58286 | 572 | 29 | 543 |
| 4 | JKiwi | 0.91 | 0.95 | Java | 8851 | 45 | 23 | 22 |
| 5 | Joda | 1.0 | 2.1 | Java | 34705 | 135 | 103 | 32 |
| 6 | JStock | 1.0.5 | 1.0.6 | Java | 35205 | 207 | 108 | 99 |
| 7 | JText | 5.0 | 5.1 | Java | 67875 | 314 | 181 | 133 |
| 8 | LWJGL | 1.0 | 2.8 | Java | 2813 | 31 | 26 | 5 |
| 9 | ModBus | 1.01 | 1.02 | Java | 4212 | 86 | 69 | 17 |
| 10 | OpenGTS | 2.1.6 | 2.4.0 | Java | 60593 | 161 | 131 | 30 |
| 11 | OpenRocket | 1.1.6 | 12.03 | Java | 9279 | 83 | 34 | 49 |
| 12 | Quartz | 1.5.2 | 1.6.6 | Java | 19123 | 93 | 83 | 10 |
| 13 | Spring | 1.2 | 1.2.9 | Java | 111665 | 1333 | 588 | 745 |
| 14 | SubSonic | 2.8 | 4.6 | Java | 9162 | 121 | 95 | 26 |

*Total Change* $=$ *No. of lines added* $+$ *No. of lines deleted* $+$ $2 *$ *No. of lines modified.*

After calculating the exact changes for each class, they are examined for odors of code smell using a commercial tool called Understand (http://www.scitools.com). It is mandatory to estimate metric values for each system and export the results in a comma-separated-value list. Threshold values selected for metrics are then applied and marked for the truth value of each smell in each class. The thresholds used for all metrics are along with the corresponding smell.

### B. MACHINE LEARNING METHODS

#### 1) RANDOM FOREST

It is an ensemble classifying approach which comprises of various decision trees. In each class, separate tree yields out various modes of the class as outputs. Random forests are collections of trees with all slightly varied. It randomizes the algorithm, not the training data. In this research, for each of the software systems, a random forest of 10 trees has been constructed while considering 4 random independent variables at each node. Precise forecast of blame inclined modules in programming improvement process empowers successful disclosure and recognizable proof of the deformities. Such prediction models are particularly significant for the vast scale frameworks, where check specialists need to center their consideration and assets to issue ranges in the framework being worked on. It is used to introduce a system for anticipating flaw by enhancing and grouping precision by growing an outfit of trees and giving them a chance to vote on the order choice. The datasets used in this paper differ in estimate, yet all normally contain few deformity tests. On the off chance that general exactness augmentation is the objective, at that point gaining from such information more often than not brings about a one-sided classifier. To acquire better expectation of blame inclination, several procedures are examined: appropriate inspecting strategy in building the tree classifiers, and limit alteration in deciding the triumphant class. Finally, it is observed with a powerful precision.

When different versions of software are launched the metrics of a software changes and a new version of software is aimed at fault negation. As the metrics are changing, there might be appearance of new faults. Thus, we also need to observe the accuracy our trained model reaches when faults are corrected. After correction of the faults we need to predict the new faults or changes that may arise. Moreover, to train our model sufficient amount of data of different modules is required so that our model can reach the desired prediction accuracy. Indeed, considering these scenarios it is important to train the classifiers on several versions of the same software.

### 2) NAÏVE BAYES

This comprises of a normal probabilistic classifier that relies on the application of Bayes' theorem having strong independence presumptions between features. Naïve Bayes is utilized to adjust classes with and without deserts and to choose most essential measurements from all accessible, as some of them ought to have apparently little effect on the nearness of genuine programming deserts. Consequences of testing mixes of above machine learning components for best forecast outcomes are displayed utilizing different datasets – with and without code terrible stenches. Expectation models were watched for dataset with code terrible stenches utilized, when Naïve Bayes calculation is utilized. It likewise gives ideal subset for preparing and assessment of Random Forest classifier. Furthermore, such blend was chosen for definite assessment of use of code smells-based measurements in deformity forecast process.

### 3) BAGGING

Bagging stands for bootstrap aggregation algorithm which is also an ensemble method for machine learning. Bootstrapping is the method for selection of samples from the original population which are used for estimation of various statistics or model accuracy. It lessens difference and keeps away from over-fitting. In spite of the fact that it is normally connected to tree techniques, it can be utilized with a strategy. Sacking produces a few diverse preparing sets of similar sizes with substitution, and after that fabricates a model by voting in favor of an ostensible target or averaging for a numeric target

### 4) DECISION TREE

In Decision Tree, a free factor is chosen at every hub of the tree. The tree is crossed amid characterization till the point when a leaf hub is received. Each leaf hub correlates with a choice or grouping. ID3 calculation was utilized to construct the choice tree in this research. Designers constantly keep up programming frameworks to adjust to new prerequisites and to settle bugs. Because of the unpredictability of support assignments and an opportunity to-advertise, engineers settle on poor execution decisions, otherwise called code smells. It is widely known that code smells obstruct fathom ability, and potentially increment change-and blame inclination. Hence, they should be distinguished to empower the use of rectifications.

Wrong meanings of code smells influence engineers to differ whether a bit of code is a scent or not; thus making troublesome production of an all-inclusive discovery arrangement ready to perceive smells in various programming ventures. A few works have been proposed to recognize code smells, however regardless they reported wrong outcomes. In this research, we contemplate the adequacy of the Decision Tree calculation to perceive code smells. For this, it was connected to a dataset containing several open source ventures and the outcomes were contrasted and the manual prophet, with existing recognition approaches and other machine learning calculations. The outcomes demonstrated that the approach could viably learn rules for the identification of the code smells contemplated. The outcomes were better when hereditary calculations are utilized to pre-select measurements.

### 5) LOGITBOOST

LogitBoost is a boosting algorithm which involves utilization of a generalized model (additive in nature) and then applies the $f(cost)$ of logistic regression to derive the LogitBoost. For programming change, openness of benefits is compelled, in this way requiring profitable & potent usage of advantages. It might be proficient through desire of key characteristics, which impact programming quality, for instance, accuse slant, change slant, effort, reasonableness, et cetera. Predicting the classes that are slanted to changes may assist in testing. This may decrease the costs related with programming bolster. The examination exhibits that machine-learning techniques are more capable than backslide strategies. Among the machine-learning systems, boosting procedure (i.e. Logitboost) beat the different models. Along these lines, the model can be used to anticipate the change slant of classes, inciting upgraded programming quality.

### 6) MULTILAYER PERCEPTRON

In reality, software is very random there can be any number of modules and different types of modules having different implementations. Our model is only valid for the software for which we have trained the model, our model will be able to determine the change and the faults for the software we have trained the model for and the also the upcoming versions. It cannot predict for different softwares but for software having similar metrics and modules. For example, Google has developed two word-embedding for Word2Vec: one is for data taken from newspapers and another is for data taken for Twitter. We have done a study to test the accuracy of the LSTM neural network on these types of software data and for that we take 14 different software and results suggest that it has outperformed other algorithms previously used.

Multilayer Perceptron is a class of feed forward artificial neural network. For this study, only 1 hidden layer was used and there was only 1 output node in output layer whose value is greater than a threshold (cut-off point) to show whether the class was undergoing a change or not. Programming flaw in the present time is most basic in the field of program building. Most of the affiliations used diverse procedures to foresee abandons in their things already that are passed on. Deformation desire frameworks help the relationship to use their advantages enough which achieves bring down cost and time necessities. There are distinctive systems that are used for foreseeing surrenders in programming before it must be passed on. In this paper, Multilayer Perceptron is used. This procedure completed on different stages and the results are diverged from find which figuring conveys better results.

## IV. RESULTS AND DISCUSSION

The changes predicted using six machine learning algorithms with the help of code smell are compared and analyzed against the actual changes that occurred between the two versions of the 14 software described in Table 1. This section analyses the effectiveness of code smell in predicting whether a class will undergo change in the subsequent versions or not. For performance evaluation, the following measures are used:

1) *Sensitivity:* The sensitivity is the percentage of the correctly predicted classes in the error category.
2) *Specificity:* The specificity is the level of the accurately anticipated classes in the no-error classification; it is the supplement of the error rate, for a whole scope of Cutoff points.
3) *Cutoff Points:* Cutoff points are used to get a congruity between the measure of classes expected as change slanted.
4) *AUC (Area Under Curve):* It is defined as the desire that a consistently drawn irregular positive is positioned before a consistently drawn arbitrary negative. Various examinations have exactly approved the connection between object oriented metrics and imperative external attributes, for example, reliability, effort, fault proneness, change proneness, and so forth. Since this paper has investigated various issues of change proneness, and knowing the fact that software experiences number of changes for the duration of its life cycle - to enhance usefulness, to settle bugs, to include new highlights and so on. Also, prerequisites of the client may change with time, prompting further changes in the software. It might bring about different adaptations of software. It may be conceivable that a solitary change in a class is proliferated to different classes, which will prompt change in the classes influenced by the change. As a result, area under the curve that signifies the largest percentage is defined to know the classes which are prone to changes. This will help us to focus on this change prone classes & make more flexible software by modifying the classes which are more prone to changes.

The experimental results for each machine learning method are described below.

### A. RANDOM FOREST

For each of the software systems, a random forest of 10 trees was constructed and each constructed while considering 4 random independent variables at each node. Table 2 and Figure 1 indicate the 10-cross validation results of the 14 software systems.

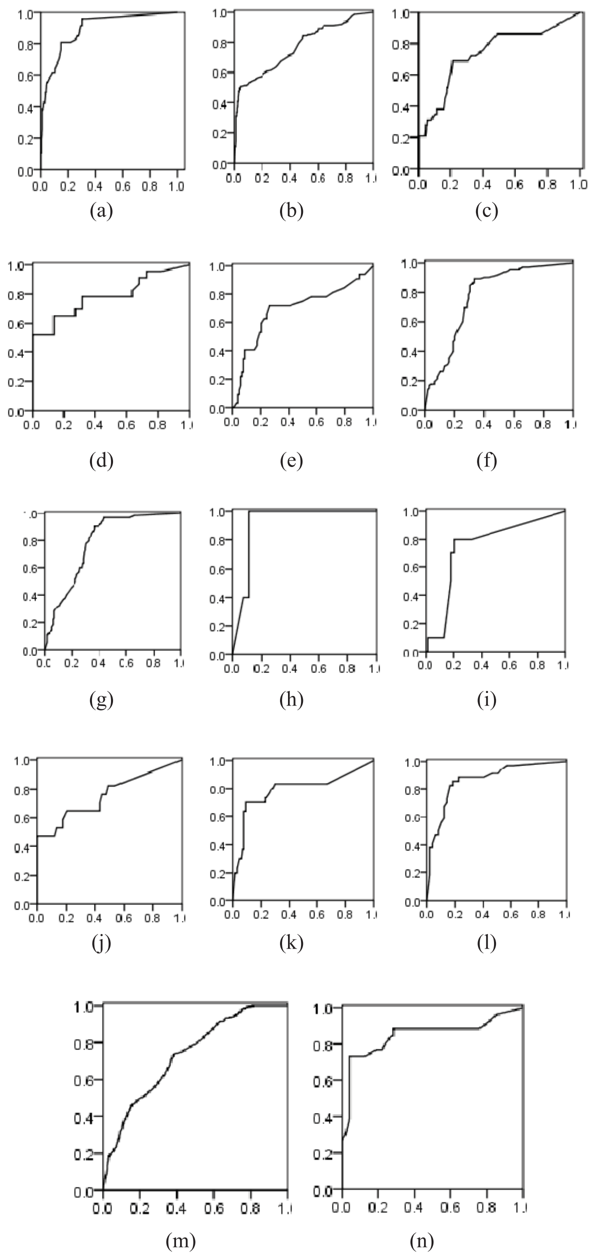

**FIGURE 1.** **ROC curves obtained for Random Forest Analysis. (a) AOI. (b) CheckStyle. (c) FreePlane. (d) jKiwi. (e) Joda. (f) jStock. (g) jText. (h) LWJGL. (i) Quartz. (j) ModBus. (k) OpenGTS. (l) OpenRocket. (m) Spring and (n) SubSonic software.**

**TABLE 2.** 10-Cross validation results for Random Forest.

| | Sensitivity | Specificity | Cutoff Point | AUC |
|---|---|---|---|---|
| AOI | 0.809 | 0.851 | 0.301500 | 0.893 |
| CheckStyle | 0.531 | 0.889 | 0.187000 | 0.769 |
| FreePlane | 0.724 | 0.670 | 0.034000 | 0.740 |
| JKiwi | 0.739 | 0.682 | 0.346000 | 0.786 |
| Joda | 0.719 | 0.738 | 0.162000 | 0.691 |
| JStock | 0.889 | 0.667 | 0.350000 | 0.777 |
| JText | 0.915 | 0.602 | 0.379500 | 0.786 |
| LWJGL | 1.000 | 0.885 | 0.226500 | 0.915 |
| ModBus | 0.647 | 0.797 | 0.251000 | 0.756 |
| OpenGTS | 0.700 | 0.908 | 0.425000 | 0.790 |
| OpenRocket | 0.824 | 0.837 | 0.622000 | 0.866 |
| Quartz | 0.800 | 0.795 | 0.143000 | 0.747 |
| Spring | 0.701 | 0.636 | 0.560500 | 0.732 |

**TABLE 3.** 10-cross validation results for random forest with 14 software systems.

| | Sensitivity | Specificity | Cutoff Point | AUC |
|---|---|---|---|---|
| AOI | 0.830 | 0.856 | 0.345000 | 0.902 |
| CheckStyle | 0.538 | 0.863 | 0.185000 | 0.757 |
| FreePlane | 0.759 | 0.746 | 0.043000 | 0.784 |
| JKiwi | 0.826 | 0.727 | 0.259000 | 0.834 |
| Joda | 0.813 | 0.641 | 0.156000 | 0.738 |
| JStock | 0.889 | 0.667 | 0.338500 | 0.758 |
| JText | 0.931 | 0.564 | 0.531500 | 0.703 |
| LWJGL | 1.000 | 0.885 | 0.292500 | 0.912 |
| ModBus | 0.765 | 0.754 | 0.158000 | 0.690 |
| OpenGTS | 0.767 | 0.931 | 0.403500 | 0.836 |
| OpenRocket | 0.824 | 0.816 | 0.521500 | 0.874 |
| Quartz | 0.900 | 0.687 | 0.735000 | 0.796 |
| Spring | 0.650 | 0.639 | 0.552500 | 0.715 |
| SubSonic | 0.731 | 0.937 | 0.586000 | 0.862 |

### B. NAÏVE BAYES

Table 3 and Figure 2 indicate the 10-cross validation results of Naïve Bayes classifier for each of the 14 software systems under study.

### C. BAGGING

Table 4 and Figure 3 indicate the 10-cross validation result of bagging obtained for each of the 14 software systems.

**TABLE 4.** 10-cross validation results for Bagging.

| | Sensitivity | Specificity | Cutoff Point | AUC |
|---|---|---|---|---|
| AOI | 0.830 | 0.837 | 0.287500 | 0.903 |
| CheckStyle | 0.510 | 0.949 | 0.221500 | 0.750 |
| FreePlane | 0.517 | 0.836 | 0.048500 | 0.689 |
| JKiwi | 0.826 | 0.818 | 0.378000 | 0.821 |
| Joda | 0.688 | 0.689 | 0.201500 | 0.711 |
| JStock | 0.907 | 0.687 | 0.329500 | 0.766 |
| JText | 0.931 | 0.591 | 0.514000 | 0.759 |
| LWJGL | 1.000 | 0.985 | 0.145000 | 0.908 |
| ModBus | 0.647 | 0.783 | 0.150000 | 0.655 |
| OpenGTS | 0.767 | 0.931 | 0.346500 | 0.799 |
| OpenRocket | 0.882 | 0.857 | 0.504500 | 0.833 |
| Quartz | 0.400 | 0.783 | 0.112000 | 0.543 |
| Spring | 0.723 | 0.628 | 0.520500 | 0.732 |
| SubSonic | 0.731 | 0.937 | 0.383000 | 0.791 |

### D. DECISION TREE

Table 5 and Figure 4 indicate the 10-cross validation results obtained for each of 14 systems.

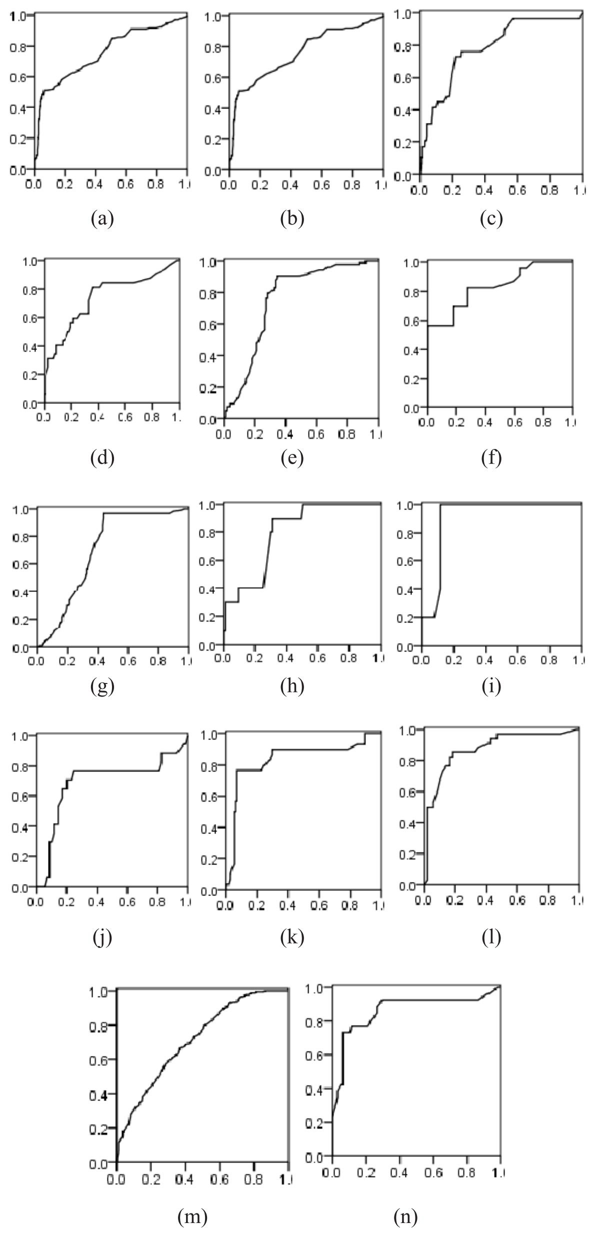

**FIGURE 2.** ROC curves for Naive Bayes Analysis. (a) AOI. (b) CheckStyle. (c) FreePlane. (d) jKiwi. (e) Joda. (f) jStock. (g) jText. (h) LWJGL. (i) Quartz. (j) ModBus. (k) openGTS. (l) OpenRocket. (m) Spring and (n) SubSonic software.

### E. LOGITBOOST

Table 6 and Figure 5 indicate the 10-cross validation results for each of 14 software systems.

### F. MULTILAYER PERCEPTRON

The result of 10-cross validation obtained over multilayer perceptron technique over the data indicated in Table 7 and Figure 6.

### G. MODEL EVALUATION

Any self-assured cut off point is not selected to get conformity between the measure of classes expected as change slanted.

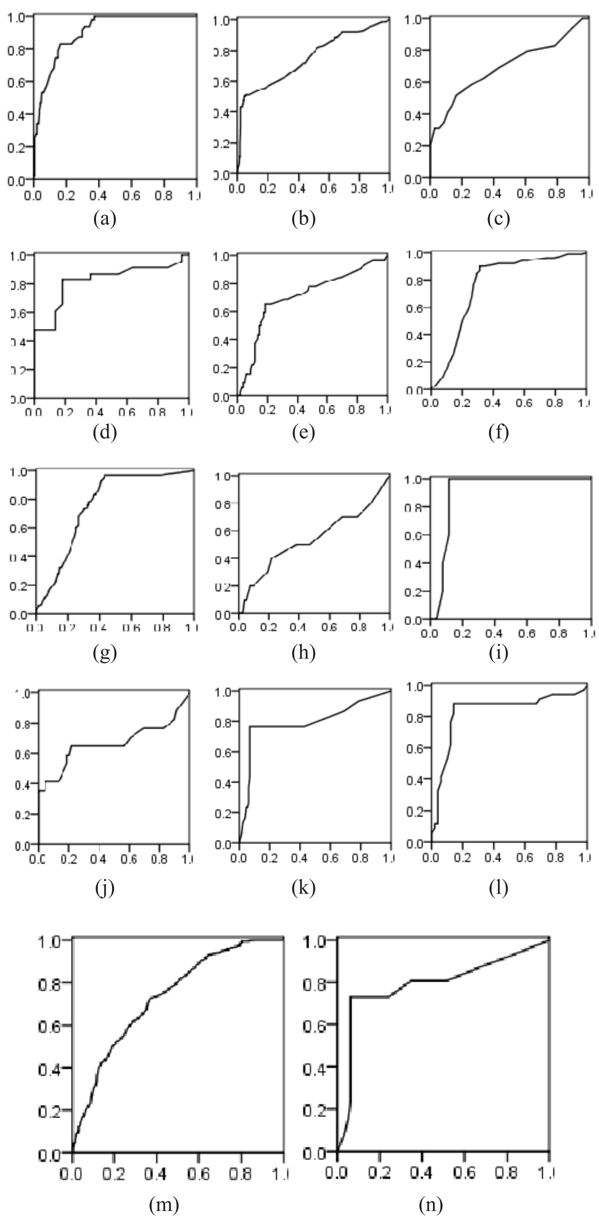
FIGURE 3. ROC curves for Bagging Analysis. (a) AOI. (b) CheckStyle. (c) FreePlane. (d) jKiwi. (e) Joda. (f) jStock. (g) jText. (h) LWJGL. (i) Quartz. (j) ModBus. (k) OpenGTS. (l) openRocket. (m) Spring and (n) SubSonic software.

The cut-off motivation driving the need indicate was directed by ROC examination. The AUC metric is used for enrolling the precision of the ordinary models. The AUC of the models expected utilizing Multilayer Perceptron is more obvious than that of the others. Both the affectability and specificity ought to be high to envision unimaginable and awful areas. Overall, in terms of sensitivity, specificity and AUC, the best model suitable for predicting a class in change prone or not is determined as Multilayer Perceptron.

For a technique to be effective in making predictions, a probability of correct classification should be at least 70%. After software is developed, there is a number of changes and faults that start cropping up in the working and functioning of the software. The main aim of a Machine Learning or a Deep Learning system is to determine in which modules there can be changes in functioning of the software (that is in the maintenance phase of software). By considering the metrics

TABLE 5. 10-cross validation results for Decision Tree.

| | Sensitivity | Specificity | Cutoff Point | AUC |
|---|---|---|---|---|
| AOI | 0.809 | 0.856 | 0.296000 | 0.889 |
| CheckStyle | 0.510 | 0.878 | 0.185000 | 0.752 |
| FreePlane | 0.586 | 0.820 | 0.091500 | 0.741 |
| JKiwi | 0.654 | 0.864 | 0.583500 | 0.802 |
| Joda | 0.699 | 0.687 | 0.882000 | 0.650 |
| JStock | 0.889 | 0.697 | 0.370500 | 0.752 |
| JText | 0.923 | 0.619 | 0.387500 | 0.790 |
| LWJGL | 1.000 | 0.885 | 0.250000 | 0.904 |
| ModBus | 0.647 | 0.739 | 0.225000 | 0.743 |
| OpenGTS | 0.700 | 0.924 | 0.550000 | 0.766 |
| OpenRocket | 0.853 | 0.857 | 0.568000 | 0.858 |
| Quartz | 0.600 | 0.819 | 0.198500 | 0.658 |
| Spring | 0.622 | 0.725 | 0.438000 | 0.731 |
| SubSonic | 0.731 | 0.937 | 0.471000 | 0.840 |

TABLE 6. 10-cross validation results for LogitBoost.

| | Sensitivity | Specificity | Cutoff Point | AUC |
|---|---|---|---|---|
| AOI | 0.809 | 0.847 | 0.318000 | 0.907 |
| CheckStyle | 0.531 | 0.876 | 0.175500 | 0.756 |
| FreePlane | 0.724 | 0.790 | 0.085500 | 0.759 |
| JKiwi | 0.783 | 0.682 | 0.232500 | 0.848 |
| Joda | 0.813 | 0.660 | 0.123000 | 0.752 |
| JStock | 0.907 | 0.677 | 0.442500 | 0.749 |
| JText | 0.923 | 0.409 | 0.473000 | 0.756 |
| LWJGL | 1.000 | 0.885 | 0.254500 | 0.900 |
| ModBus | 0.647 | 0.174 | 0.249000 | 0.646 |
| OpenGTS | 0.767 | 0.901 | 0.209500 | 0.833 |
| OpenRocket | 0.794 | 0.878 | 0.676500 | 0.878 |
| Quartz | 0.900 | 0.819 | 0.135000 | 0.797 |
| Spring | 0.619 | 0.704 | 0.604500 | 0.722 |
| SubSonic | 0.731 | 0.937 | 0.229500 | 0.866 |

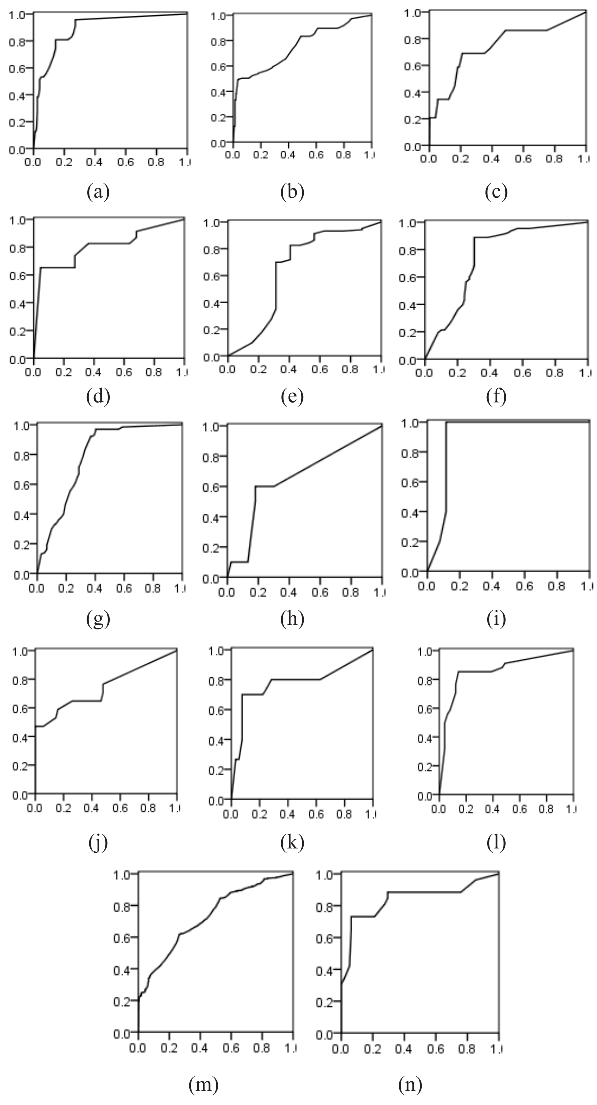
FIGURE 4. **ROC curves for Decision Tree Analysis. (a) AOI. (b) CheckStyle. (c) FreePlane. (d) jKiwi. (e) Joda. (f) jStock. (g) jText. (h) LWJGL. (i) Quartz. (j) ModBus. (k) OpenGTS. (l) OpenRocket. (m) Spring and (n) SubSonic software.**

TABLE 7. **10-cross validation results for Multilayer Perceptron.**

| | Sensitivity | Specificity | Cutoff Point | AUC |
|---|---|---|---|---|
| AOI | 0.830 | 0.748 | 0.213000 | 0.888 |
| CheckStyle | 0.531 | 0.870 | 0.189500 | 0.767 |
| FreePlane | 0.724 | 0.807 | 0.051000 | 0.790 |
| JKiwi | 0.609 | 0.955 | 0.738500 | 0.790 |
| Joda | 0.750 | 0.689 | 0.103500 | 0.725 |
| JStock | 0.889 | 0.677 | 0.424000 | 0.757 |
| JText | 0.900 | 0.624 | 0.439000 | 0.778 |
| LWJGL | 1.000 | 0.885 | 0.277500 | 0.896 |
| ModBus | 0.588 | 0.884 | 0.308000 | 0.749 |
| OpenGTS | 0.700 | 0.916 | 0.507000 | 0.827 |
| OpenRocket | 0.853 | 0.857 | 0.548000 | 0.847 |
| Quartz | 0.800 | 0.819 | 0.169000 | 0.797 |
| Spring | 0.650 | 0.651 | 0.552000 | 0.728 |
| SubSonic | 0.731 | 0.947 | 0.516000 | 0.852 |

of different modules of software, it can be predicted that in the future, when the software is into functioning what faults may arise and in which modules. 70% chance to predict the amount of change means that the Machine Learning system can predict 70% of the changes that can occur in the future when the software is used. Here, two different measures are used to evaluate the correctness of models: sensitivity and specificity. As we know the number of classified true instances which are absolutely correct is equal to the number of classified false instances which are absolutely correct. For a model to be effective, both these values must be high. This would mean that the model makes correct classifications for both true and false values. In other words, the model performs well for both true and false values.

From the results, it is clear that Multilayer Perceptron performs best in comparison to all other models. In our dataset of 14 software systems, it exhibited a sensitivity superior to 0.70 or in most cases and specificity higher than 0.67 or more. This means that over the dataset of 4120 classes, the multilayer perceptron was able to correctly classify 1272 change prone classes out of 1817 and 1544 classes as not change prone out of 2303. This encourages results in Predicting Class Change Proneness.

Any self-assured cut off point is not selected to get conformity between the measure of classes expected as change slanted. The cut-off motivation driving the need indicate was directed by ROC examination. The AUC metric is used for enrolling the precision of the ordinary models. The AUC of the models expected utilizing Multilayer Perceptron is more obvious than that of the others. Both the affectability and specificity ought to be high to envision unimaginable and awful areas. Overall, in terms of sensitivity, specificity and AUC, the best model suitable for predicting a class in change prone or not is determined as Multilayer Perceptron.

For a technique to be effective in making predictions, a probability of correct classification should be at least 70%. After software is developed, there is a number of changes and faults that start cropping up in the working and functioning of the software. The main aim of a Machine Learning or a

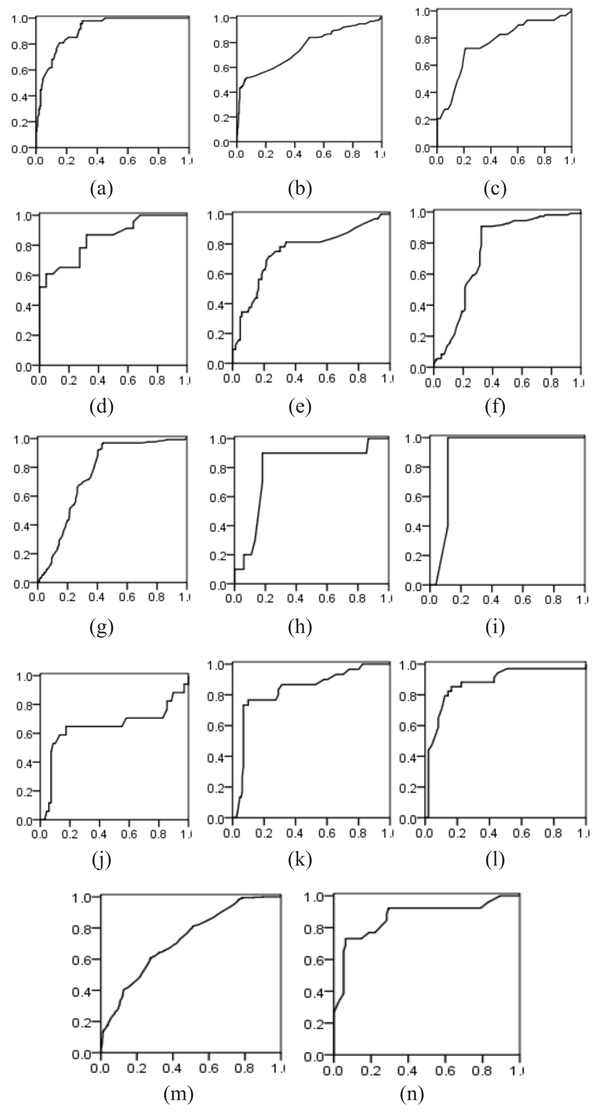
FIGURE 5. ROC curves for Logit Boost Analysis. (a) AOI. (b) CheckStyle. (c) FreePlane. (d) jKiwi. (e) Joda. (f) jStock. (g) jText. (h) LWJGL. (i) Quartz. (j) ModBus. (k) OpenGTS. (l) OpenRocket. (m) Spring and (n) SubSonic software.

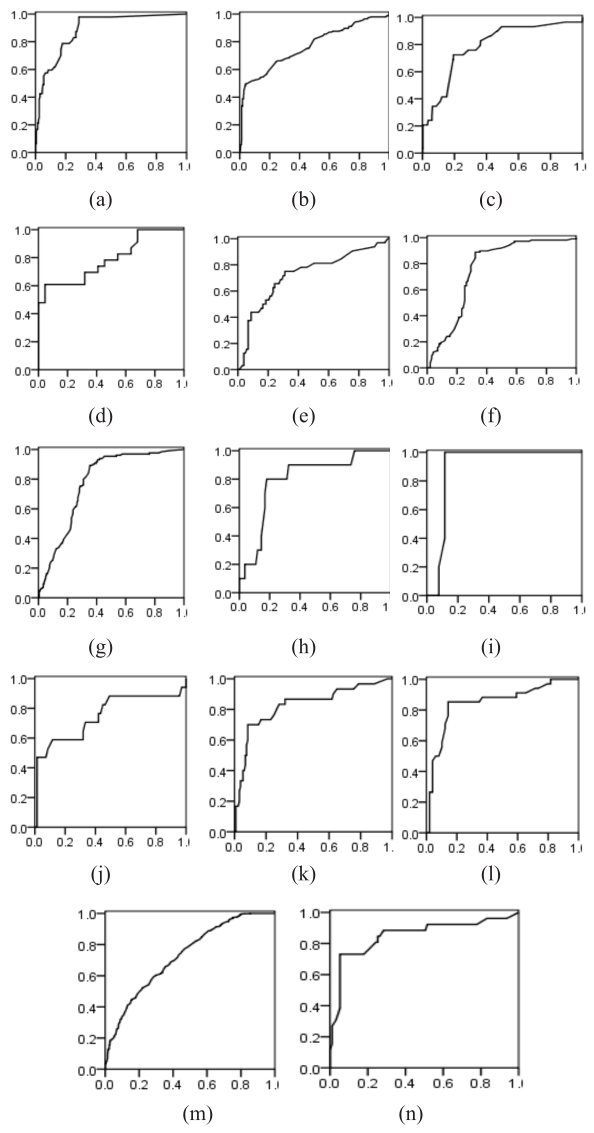
FIGURE 6. ROC curves for Multilayer Perceptron Analysis. (a) AOI. (b) CheckStyle. (c) FreePlane. (d) jKiwi. (e) Joda. (f) jStock. (g) jText. (h) LWJGL. (i) Quartz. (j) ModBus. (k) openGTS. (l) openRocket. (m) Spring and (n) SubSonic software.

Deep Learning system is to determine in which modules there can be changes in functioning of the software (that is in the maintenance phase of software). By considering the metrics of different modules of software, it can be predicted that in the future, when the software is into functioning what faults may arise and in which modules. 70% chance to predict the amount of change means that the Machine Learning system can predict 70% of the changes that can occur in the future when the software is used. Here, two different measures are used to evaluate the correctness of models: sensitivity and specificity. As we know the number of classified true instances which are absolutely correct is equal to the number of classified false instances which are absolutely correct. For a model to be effective, both these values must be high. This would mean that the model makes correct classifications for both true and false values. In other words, the model performs well for both true and false values.

From the results, it is clear that Multilayer Perceptron performs best in comparison to all other models. In our dataset of 14 software systems, it exhibited a sensitivity superior to 0.70 or in most cases and specificity higher than 0.67 or more. This means that over the dataset of 4120 classes, the multilayer perceptron was able to correctly classify 1272 change prone classes out of 1817 and 1544 classes as not change prone out of 2303. This encourages results in Predicting Class Change Proneness.

### H. PREDICTION

The predictions obtained from the algorithms also help in detecting the Code smells as the faults or change is related to code smells. The pre-identification of smells checks accuracy of the learning algorithms. The algorithm tested on a software with minimum code smell gives better accuracy than a software with code smell. Thus, decreasing in the accuracy of an algorithm is an indication of code smell and also caps the performance of the algorithm.

**How Is The Prediction Useful for a Software Developer?**

From Machine Learning perspective, a dataset is divided into two parts: a training dataset and a test dataset. An algorithm is trained to the different outputs and the prediction ability of an algorithm is tested based on the test dataset i.e. whether the algorithm has been able to learn properly or not. In our scenario, the dataset contains the metrics of a specific module in different software, and the outputs are the faults and changes that have come up. Indeed, we have divided the dataset into training and test and applied our neural network on the training dataset by which our model is able to learn how to predict changes in software. The model is then tested on the test dataset and the accuracy is determined i.e. given the metrics of the module whether our model is able to give proper prediction of the faults.

When software is developed, it is very hard for a software developer or a team of software developers to understand the look and feel of the software or how it is working as unlike other engineering objects it is not physically present. For this reason, there can be a number of errors that may arise in the software when it is used by the customer or the client. There might be errors that might have not come up during the testing phase. In such a situation, any error discovered during the maintenance phase is challenging to handle like any unforsaken situation. Our model is a very handy tool for an organization or a software developer as it will raise an alarm to the software developer pointing out the models that might be at fault and the errors that might occur after software is developed and handed over to the client.

### *I. COMPARISON OF OUR RESEARCH WITH THE PREVIOUS WORKS*

In predicting class change proneness, the empirical evidence in [3] and [8] is still limited because of small number of datasets. Subsequently the probability of smell detectors becomes less. Past works analyzed the impact of code smells on change and accuse slant, relied upon data got from customized see pointers. But such discoveries are frequently prepared to achieve a conventional level of precision; it has been up till now that their innate imprecision impacts the outcomes of the examination. The previous works demonstrated that a few smells can be more destructive than others, however their examination did not think about the greatness of watched wonder. This was illustrated in the experiments of this research appropriately. Various studies have been made to indicate that classes influenced by code smells have more opportunities to display abandons than other classes; however, using huge datasets and applying multiple machine learning algorithms such as in this research, this observation has been observed more clearly with changes and defects being significantly low.

## V. CONCLUSIONS

The aim of this paper is to determine prediction power for class level change proneness through code smell. Firstly, a total of 4120 classes were selected for this study after preprocessing. Two versions of 14 softwares were compared to find out the change prone classes. After that, a ten-cross validation was used to divide the dataset into ten subsets among which nine for training machine learning model. Six machine learning algorithms (Naive Bayes Classifier, Multilayer Perceptron, LogitBoost, Bagging, Random Forest, and Decision Tree) were used to predict change proneness using code smells. The predicted values were then compared and analyzed against the actual Change Prone classes using ROC analysis to get the false positives and true positives. It has been concluded from this study that: a) Code smell can predict class change proneness with a probability superior to 70%, b) the Multilayer Perceptron provides the best results and prediction power in comparison or other machine learning models.

Even though we have performed extensive analysis as above, there are some limitations regarding the quantity of code smells and adaptation to a specific context. Therefore, in the future, we aim to provide solution to some problems that might occur: Firstly, we need to design a suitable threshold technique which is mathematically well-posed [23]–[29]. Secondly, the number of code smells should be studied to enhance the performance of prediction.

## ACKNOWLEDGMENTS

The authors would like to thank the Editor-in-Chief and the anonymous reviewers for their valuable comments and suggestions.

## REFERENCES

[1] M. Fowler, *Refactoring—Improving the Design of Existing Code*, 1st ed. Boston, MA, USA: Addison-Wesley, 1999.

[2] G. Rasool and Z. Arshad, "A lightweight approach for detection of code smells," *Arabian J. Sci. Eng.*, vol. 42, no. 2, pp. 483–506, Feb. 2017.

[3] M. Gatrell and S. Counsell, "The effect of refactoring on change and fault-proneness in commercial C# software," *Sci. Comput. Program.*, vol. 102, pp. 44–56, May 2015.

[4] G. Hecht, N. Moha, and R. Rouvoy, "An empirical study of the performance impacts of android code smell," in *Proc. Int. Workshop Mobile Softw. Eng. Syst.*, May 2016, pp. 59–69.

[5] N. Sae?Lim, S. Hayashi, and M. Saeki, "Context-based approach to prioritize code smells for prefactoring," *J. Softw. Evol. Process*, vol. 30, no. 6, p. e1886, Jun. 2017. doi: 10.1002/smr.1886.

[6] A. Kaur, K. Kaur, and S. Jain, "Predicting software change-proneness with code smells and class imbalance learning," in *Proc. IEEE Int. Conf. Adv. Comput., Commun. Inform. (ICACCI)*, Sep. 2016, pp. 746–754.

[7] K. Kaur and S. Jain, "Evaluation of Machine Learning Approaches for Change-Proneness Prediction Using Code Smell," in *Proc. 5th Int. Conf. Frontiers Intell. Comput. Theory Appl.*, Mar. 2017, pp. 561–572.

[8] F. Khomh, M. Penta, and Y. Gueheneuc, "An exploratory study of the impact of code smells on software change-proneness," *Proc. 16th Working Conf. Reverse Eng.*, Oct. 2009, pp. 75–84.

[9] W. Ma, L. Chen, Y. Zhou, and B. Xu, "Do we have a chance to fix bugs when refactoring code smells?" *Proc. IEEE Int. Conf. Softw. Anal. Test. Evol. (SATE)*, Nov. 2016, pp. 24–29.

[10] R. Morales, Z. Soh, F. Khomh, G. Antoniol, and F. Chicano, "On the use of developers' context for automatic refactoring of software anti-patterns," *J. Syst. Softw.*, vol. 128, pp. 236–251, Jun. 2017.

[11] R. Nascimento and C. Sant'Anna, C.: "Investigating the relationship between bad smell and bugs in software systems," in *Proc. 11th Brazilian Symp. Softw. Compon., Archit. Reuse*, Sep. 2017, p. 4.

[12] F. Palomba, N. D. Di, A. Panichella, A. Zaidman, and A. De Lucia, "Lightweight detection of Android-specific code smell: The aDoctorproject," in *Proc. IEEE 24th Int. Conf. Softw. Anal., Evol. Reeng. (SANER)*, Feb. 2017, pp. 487–491.

[13] F. Palomba, M. Zanoni, F. A. Fontana, A. D. Lucia, and R. Oliveto, "Smell like teen spirit: Improving bug prediction performance using the intensity of code smell," in *Proc. IEEE Int. Conf. Softw. Maintenance Evol. (ICSME)*, Oct. 2016, pp. 244–255.

[14] S. S. Rathore and S. Kumar, "Towards an ensemble based system for predicting the number of software faults," *Expert Syst. Appl.*, vol. 82, pp. 357–382, Oct. 2017.

[15] A. Saboury, P. Musavi, F. Khomh, and G. Antoniol, "An empirical study of code smell in JavaScript projects," in *Proc. IEEE 24th Int. Conf. Softw. Anal., Evol. Reeng. (SANER)*, Feb. 2017, pp. 294–305.

[16] S. A. Vidal, C. Marcos, and J. A. DÃČÂaz-Pace, "An approach to prioritize code smells for refactoring," *Automated Softw. Eng.*, vol. 23, no. 3, pp. 501–532, Sep. 2016.

[17] M. Hadj-Kacem and N. Bouassida, "A hybrid approach to detect code smells using deep learning," in *Proc. 13th Int. Conf. Eval. Novel Approaches Softw. Eng. (ENASE)*, Madeira, Portugal, Mar. 2018.

[18] F. Fontana, M. Zanoni, A. Marino, and M. Mantyla, "Code smell detection: Towards a machine learning-based approach," in *Proc. IEEE Int. Conf. Softw. Maintenance*, Sep. 2013, pp. 1–5.

[19] U. Azadi, F. Fontana, and M. Zanoni, "Machine learning based code smell detection through WekaNose," in *Proc. 40th Int. Conf. Softw. Eng. Companion*, May 2018, pp. 288–289.

[20] D. Nucci, F. Palomba, D. Tamburri, A. Serebrenik, and A. D. Lucia, "Detecting code smells using machine learning techniques: Are we there yet?" in *Proc. IEEE 25th Int. Conf. Softw. Anal., Evolution Reeng. (SANER)*, Campobasso, Italy, Mar. 2018, pp. 612–621.

[21] M. Z. Iqbal and S. Sherin, "Empirical studies omit reporting necessary details: A systematic literature review of reporting quality in model based testing," *Comput. Standards Interfaces.*, vol. 55, pp. 156–170, Jan. 2018.

[22] L. Kumar, S. Misra, and S. K. Rath, "An empirical analysis of the effectiveness of software metrics and fault prediction model for identifying faulty classes," *Comput. Standards Inter.*, vol. 53, pp. 1–3, Apr. 2017.

[23] S. Jha *et al.*, "Neutrosophic soft set decision making for stock trending analysis," *Evolving Syst.*, to be published. doi: 10.1007/s12530-018-9247-7.

[24] N. T. Thong, L. Q. Dat, L. H. Son, N. D. Hoa, and M. Ali, "Florentin smarandache, dynamic interval valued neutrosophic set: Modeling decision making in dynamic environments," *Comput. Ind.*, to be published.

[25] M. Ali, L. Q. Dat, L. H. Son, and F. Smarandache, "Interval complex neutrosophic set: Formulation and applications in decision-making," *Int. J. Fuzzy Syst.*, vol. 20, no. 3, pp. 986–999, 2018.

[26] R. T. Ngan, L. H. Son, B. C. Cuong, and M. Ali, "H-max distance measure of intuitionistic fuzzy sets in decision making," *Appl. Soft Comput.*, vol. 69, pp. 393–425, Aug. 2018.

[27] M. Khan, L. H. Son, M. Ali, H. T. M. Chau, N. T. N. Na, and F. Smarandache, "Systematic review of decision making algorithms in extended neutrosophic sets," *Symmetry-Basel*, vol. 10, pp. 314–342, 2018.

[28] M. Ali, L. H. Son, I. Deli, and N. D. Tien, "Bipolar neutrosophic soft sets and applications in decision making," *J. Intell. Fuzzy Syst.*, vol. 33, pp. 4077–4087, Apr. 2017.

[29] T. T. Ngan, T. M. Tuan, L. H. Son, N. H. Minh, and N. Dey, "Decision making based on fuzzy aggregation operators for medical diagnosis from dental X-ray images," *J. Med. Syst.*, vol. 40, no. 12, pp. 1–7, 2016.

**NAKUL PRITAM** received the M.Tech. degree in software engineering from DTU. He has more than 5 years of experience in the software industry, have done extensive work in the areas of application security, high availability application design, and reporting. His research interests include user behavior analysis, massively scalable systems, test automation and optimization, software quality analysis, software quality improvement, and data analysis.

**MANJU KHARI** received the master's degree in information security from the Ambedkar Institute of Advanced Communication Technology and Research, India and the Ph.D. degree in computer science and engineering from the National Institute of Technology, Patna. She is currently an Assistant Professor with the Ambedkar Institute of Advanced Communication Technology and Research, under Government of NCT Delhi, affiliated with Guru Gobind Singh Indraprastha University, India. She is also a Professor In-Charge of IT services with the Ambedkar Institute of Advanced Communication Technology and Research. She has more than 12 years of experience in network planning and management. She has published 60 papers in refereed national and international journals and conferences. She has authored six book chapters and co-authored two books. Her research interests include software testing, software quality, software metrics, information security, and nature-inspired algorithms. She is a Life Member of various international and national research societies (SDIWC and IAENG). She is also a Guest Editor of the *International Journal of Advanced Intelligence Paradigms*, a Reviewer for the *International Journal of Forensic Engineering*, and an Editorial Board Member of the *International Journal of Software Engineering and Knowledge Engineering*.

**LE HOANG SON** received the Ph.D. degree in mathematics–informatics from the VNU University of Science, Vietnam National University (VNU), in 2013.

From 2007 to 2018, he was a Senior Researcher and a Vice Director of the Center for High Performance Computing, VNU University of Science, Vietnam National University. Since 2017, he has been promoted as an Associate Professor of information technology. Since 2018, he has been the Head of the Department of Multimedia and Virtual Reality, VNU Information Technology Institute, VNU. His major fields include artificial intelligence, data mining, soft computing, fuzzy computing, fuzzy recommender systems, and geographic information systems. He is a member of the Key Laboratory of Geotechnical Engineering and Artificial Intelligence, University of Transport Technology, Vietnam. He is a member of the International Association of Computer Science and Information Technology (IACSIT) and the Vietnam Society for Applications of Mathematics (Vietnam). He serves for the Editorial Board of *Applied Soft Computing* (ASOC, in SCIE), the *International Journal of Ambient Computing and Intelligence* (IJACI, in SCOPUS), and the *Vietnam Journal of Computer Science and Cybernetics* (JCC). He is an Associate Editor of the *Journal of Intelligent and Fuzzy Systems* (JIFS, in SCIE), the IEEE ACCESS (in SCIE), *Neutrosophic Sets and Systems* (NSS), *Vietnam Research and Development on Information and Communication Technology* (RD-ICT), *VNU Journal of Science: Computer Science and Communication Engineering* (JCSCE), and *Frontiers in Artificial Intelligence*.

**RAGHVENDRA KUMAR** received the B.Tech. degree in computer science and engineering from SRM University, Chennai, India, the M.Tech. degree in computer science and engineering from KIIT University, Bhubaneswar, India, and the Ph.D. degree in computer science and engineering from Jodhpur National University, Jodhpur, India. He is currently an Assistant Professor with the Computer Science and Engineering Department, L.N.C.T Group of College, Jabalpur, India. He has published 86 research papers in international/national journal and conferences including IEEE, Springer, and ACM. He has authored 12 computer science books in field of data mining, robotics, graph theory, and turing machine by IGI Global Publication, USA, IOS Press, The Netherlands, Lambert Publication, Scholar Press, Kataria Publication, Narosa, Edupedia Publication, S. Chand Publication, and Laxmi Publication. His research interests include computer networks, data mining, cloud computing and secure multiparty computations, theory of computer science, and design of algorithms. He received the Best Paper Award at the IEEE Conference 2013 and the Young Achiever Award 2016 by the IEAE Association, for his research in the field of distributed database. He serves as the Session Chair, a Co-Chair, and a Technical Program Committee Member for many international and national conferences, and a Guest Editor for many special issues from reputed journals (indexed by: Scopus, ESCI).

**SUDAN JHA** was born in Kathmandu, Nepal. He received the Proficiency in Certificate Level from the Saint Xavier's College, Kathmandu, the B.E. degree in electronics engineering from the Motilal Nehru Regional College, Allahabad, India, in 2001, and the master's degree in computer science. He was a Lecturer with the Nepal Engineering College (NEC), one of the premium and largest engineering college and the first one in the private domain in Nepal, where he got full sponsorship from the employer (NEC) for his master's study in computer science. He was promoted as an Assistant Professor of the Department of Computer Science and Engineering, and later, he became the Head of the Computer Science and Engineering Department. He chaired and organized five international conferences, some of the proceedings of those conferences have been published by Springer Verlag, World Science Series, and Imperial Press London.

**ISHAANI PRIYADARSHINI** received the B.Tech. degree in computer science engineering and the master's degree in information security from KIIT University and the master's degree in cybersecurity from the University of Delaware, USA, where she is currently pursuing the Ph.D. degree in electrical and computer engineering (cybersecurity). Her areas of interests include cryptography, network security, and machine learning.

**MOHAMED ABDEL-BASSET** received the B.Sc., M.Sc., and Ph.D. degrees in information systems and technology from the Faculty of Computers and Informatics, Zagazig University, Egypt. He focuses on the application of multi-objective and robust meta-heuristic optimization techniques. He has published more than 150 articles in international journals and conference proceedings. His current research interests include optimization, operations research, data mining, computational intelligence, applied statistics, decision support systems, robust optimization, engineering optimization, multi-objective optimization, swarm intelligence, evolutionary algorithms, and artificial neural networks. He is the Program Chair of many conferences in the fields of decision making analysis, big data, optimization, complexity, and the Internet of Things, and an Editorial Collaborator of some journals of high impact. He is an Editor/a Reviewer for different international journals and conferences.

**HOANG VIET LONG** received the Ph.D. Diploma degree in computer science from the Hanoi University of Science and Technology, in 2011. His Ph.D. Diploma dissertation was on fuzzy and soft computing field. He has been an Associate Professor of information technology, since 2017. He is currently the Head of the Faculty of Information Technology, People's Police University of Technology and Logistics, Vietnam. He is a Researcher of the Institute for Computational Science, Ton Duc Thang University, Ho Chi Minh City, Vietnam. Recently, he has been concerning in cybersecurity, machine learning, bitcoin, and block chain. He has published more than 20 papers in ISI-covered journal.