Using Code Coverage Metrics for Improving Software Defect Prediction

Bilal Al-Ahmad*

Computer Information Systems Department, Faculty of Information Technology and Systems,
The University of Jordan, AqabaBranch, Jordan

*Corresponding author; Email: b.alahmad@ju.edu.jo
Manuscript submitted September 29, 2018; accepted November 2, 2018.
doi: 10.17706/jsw.13.12.654-674

Abstract: Software defect prediction enables software developers to estimate the most defective code parts in order to reduce testing efforts. As the size of software project becomes larger, software defect prediction becomes an urgent need. While static product metrics have been extensively investi gated as a static means to predict software defects, coverage analysis of the software has been abandoned due to the expected complexities. This paper proposed a novel hybrid approach that leverages code coverage metrics to improve software defect prediction. We build and compare software defect prediction results for four distinct scenarios: static product, code coverage, hybrid, and feature selection. First scenario resembles static analysis and acts as baseline model. Second scenario addresses coverage issues of the associated test cases for the source code. Third and fourth scenarios are derived from combinations of stati c product and code coverage scenarios. Each scenario has been modeled and examined using thirteen different machine learning classifiers. Two rounds of experiments have been done. First round employs real data extracted from 23 successive releases of Apache Lucene, whereas second round applies oversampling technique for the same releases. The results indicate that code coverage scenario attains a significant improvement in software defect prediction, especially when there is a high-coverage ratio for software modules. In general, hybrid scenario outperforms the other three scenarios. Naive Bayes classifier attains the best results among all classifiers at the first round, while IBK performs well for the second round. The second round experiment exhibits a superior performance compared to the first round because it approaches two times better recall. Further, we notice a steady improvement in the latest releases of Apache Lucene project compared to the earlier ones.

Key words: Code coverage metrics, Machine learning classifier, Software defect prediction, Software quality, Static product metrics

1. Introduction

Software defects prediction is one of the most important research areas in software engineering [1]–[6]. Software development strives to improve the quality of software systems without assigning too many resources in the quality assurance activities. Software quality strongly influences the overall system performance because it involves improving the software quality by reducing the number of defects. Defects may be existed in various stages of the software development life cycle, defects detection in the later stage s of the software life cycle is considered to be complex to fix more than those found in earlier stages. The ability to predict defects is useful to create efficient resources for software testing. The earlier prediction of defects can improve testing management and quality of software.

Several studies [7]-[11] discussed the use of software defect prediction models with machine learning for various purposes in software testing process. The studies [7]–[10] used machine learning-based defect prediction models to classify each software module as “buggy” or “buggy free”. The study [11] created three defect prediction models for three different testing phases in order to monitor defects on large enterprise software.

Also, the study [12] proposed an empirical approach that integrates machine learning, statistical technique, and defect reports to support software testing management. It built predictive models for defects repair times and evaluated software testing quality over three large medical software releases. Moreover, other studies [13]–[15] focused on using the software defect data to capture the buggy software modules during testing process by using the number of residual defects in order to improve software reliability prediction. While the research study [16] proposed prediction model to estimate defects number in the recent version of software by referring to the prior stable version, such the recent version includes changes that are related to adding new metrics or fixing the defective lines of code. So, perceiving such features changes through software versions lead to decrease testing times and direct developers for developing a higher quality software.

All the current approaches overlooked the aspects of coverage analysis of the source code in their experiments and considered only the baseline product metrics: C&K metrics [17] Henderson-Sellers metrics [18], Martin metrics [19], QMOOD metrics [20], Tang metrics [21], McCabe metrics [22], and Lines of Code (LOC) metric. As a result, this paper proposes various defect prediction models that take into account the code coverage metrics which enable testing teams to manage software testing effectively.

This study investigates the significance use of coverage code metrics with regard to defect prediction. Four distinct scenarios (static product, code coverage, hybrid, and feature selection) were built by applying 13 various machine learning classifiers in order to investigate the relation between the software metrics and the defects. Subsequently, the results of constructing four different defect prediction models were analyzed. The constructing procedure utilized by applying 10 folds cross validation and oversampling techniques over 23 successive releases of Apache Lucene[1] project. Finally, two rounds of experiments were conducted on the same investigated releases in order to determine the power of combining both of static product and code coverage metrics in classifying Java classes as defective or defect free. The results of the two rounds were compared and the most important metrics were recommended with regard to defect prediction. In this research, there are four contributions that answer the following research questions:

(1) Demonstrate the value of code coverage metrics in software defect prediction, according to our experiments, hybrid scenario attains best recall values compared to the other scenarios.
RQ1. Do the code coverage metrics improve the software defect prediction?

(2) Show how machine learning classifiers exhibit an increasing performance regardless of the used scenarios, NB and IBK are the best two classifiers in our experiments.
RQ2. What are the most appropriate machine learning classifiers for software defect prediction?

(3) Prove the effectiveness of the selected sampling technique over imbalanced data as in the second round of the experiment, increasing the defective software modules leads to obtain two times better recall rather than first round.
RQ3. Does the oversampling of minority software modules improve the software defect prediction?

(4) Identify the positive behavior of the investigated releases which reflects steady improvement from the earlier releases to the later ones.
RQ4. How is the behavior of the software releases by using different evaluation measures?

[1]https://mvnrepository.com/artifact/org.apache.lucene/lucene-core

The rest of this paper is organized as follows: section 2 discusses the related work by stating the most related studies in the literature, section 3 presents the proposed methodology, section 4 shows results and discussions, and section 5 presents conclusion and perspectives.

2. Related Work

Software defect prediction plays an important role in classifying the defective modules of software projects. Many research studies in literature apply machine learning techniques using static product metrics in order to predict software defects [23]–[28]. Those metrics are classified as: C&K metrics, Henderson-Sellers metrics, Martin metrics, QMOOD metrics, Tang metrics, McCabe metrics, and Lines of Code (LOC) metric. Later on, Jureczko [24] applied an empirical analysis to measure the influences of product and process metrics upon defect prediction models.Jureczko and Madeyski [25] used Kohonens neural network and k-means clustering to determine the software projects that have the same defect prediction characteristics. All the investigated product metrics were calculated using Ckjm[2]tool.

Quite recently, Madeyski and Jureczko [29] performed an empirical assessment which includes statistical tests and multi-collinearity analysis over industrial and open source software projects. The results recommended the most significant product and process metrics in respect to software defects. Tomar andAgarwal [28] applied Weighted Least Squares Twin Support Vector Machine (WLSTSVM) for defect prediction process. In addition, the study [26] showed how the distribution of imbalance data affects the software prediction process. Consequently, the study applied an oversampling to the minority modules (i.e. defective classes) in the investigated project. The results show how the proposed algorithm achieves better prediction performance.

Further, Tang et al. [27] proposed a merged static defect analysis with software defect prediction to build software prediction model. Their study restricted only to 14 static product metrics. He et al. [2] proposed using group of simplified metrics for software defect prediction through several cases using six machine learning classifiers. The results indicated that within project defect models are able to capture higher precision than cross-projects defect models. Also, among all classifiers, Naive Bayes is the most suitable classifier to select the most useful set of simplified metrics. Similarly, Jacob et al. [23] proposed an effective framework for software defect prediction, where three feature selection techniques used to select the most important attributes that give an indication of defect existence.

All the aforementioned studies considered only the static baseline metrics and overlooked the role of code coverage metrics in software defect prediction due to complexities. This proposed approach addresses this critical limitation in order to improve the overall software quality by highlighting the coverage aspects of the source code. This proposed approach is the first approach which explores the impact of using coverage metrics software defect prediction.

3. The Proposed Methodology

In our methodology, four software defects prediction models are built using 13 machine learning classifiers that were implemented in Weka. Machine learning classifiers [30] have multiple forms: artificial neural network, linear regression, rule-based, and trees. Hence, in order to cover the diversity of the most common forms, the proposed approach uses classifiers from five different families. In particular, the selected classifiers are: (1) Bayes: Naive Bayes (NB), (2) Functions: Multi-layer Perceptron (MLP), Simple Logistic (SL), SMO, Logistic (L), (3) Lazy: IBk, KStar, LWL, (i4) Meta: AdaBoost (AB), Bagging (B), LogitBoost (LB) and(5) Trees: Decision Tree (J48), Random Forest (RF). Bayes family is a direct method that estimates

2http://gromit.iiar.pwr.wroc.pl/p_inf/ckjm the probability of the best hypothesis using Bayes theorem by building a rule based model. In the functional family, the classifier builds a function (or hypothesis) of input domain (i.e. Features) and maps it into an output of range (i.e. Labels). Lazy learners simply classify a new instance by calculating the similarity between the instances against the instances in the training set and assign the label of most similar ones to that instance.

In contrast, eager learners construct a machine learning model before testing process as a ready to use classifier models such: (1) Bayes, (2) Functions, (3) Meta, and (4) Trees families. In Meta families, the idea is to learn an expert classifier of ensemble weak classifiers combined in a way to predict a label using averaging or voting methods. Finally, in the tree family, each classifier is a form of a hierarchical tree where a node at each level represents the best attribute at that level while the arcs represent the values of that attributes.

In the training phase, the proposed approach used 10-fold cross-validation method for each experiment by repeating it 10 times for each process of building a classifier. A set of evaluation metrics were used which are precision, recall, and Receiver Operating Characteristic (ROC) area as in [31]. Precision is the ratio of relevant instances among the retrieved ones, while recall is the ratio of retrieved and relevant instances to the total amount of relevant ones. The ROC area (curve) [32] is used to check whether a classifier can separate positive and negative instances and identify the best threshold for separating them. In particular, the ROC curve is a graph plot that relates recall in function of the false positive rate on different threshold points. Each point on the ROC curve represents a recall/false positive rate pair corresponding to a particular decision threshold. Each release is used for training and testing purposes. Hence, the evaluation process is conducted on a training set of each release using 10-fold cross-validation to maintain a better accuracy for the goal of improving the performance on a particular release. The cross-validation method divides a given data into two sub-sets; training and testing sets. The building classifier uses the training set and evaluates the model using the testing set. The partitioning process on the given data is repeated several times into training and testing sets (90% for training and 10% for testing) to randomly divides the data into a form of 10 equally sized sub-sets that assures data randomness where the estimated accuracy is the average over these repetitions. The k-1 (90%) sets are used for training phase while one fold (10%) is used for testing. Consequently, the advantage of 10-fold cross-validation is inherited in that all software classes are used for both training and validation, and each class is used for validation exactly once.

This reduces the variance of the resulting estimations in increasing the value of k. In our experiments, the proposed approach sets k =10, as concluded in [33]. In contrast, the studies [4] and [25] built classifier models on a training set of initial releases while the other remaining releases of the same project are used for testing. In particular to this study, the reason behind using 10-fold cross-validation is to reduce the variance in building the classifier models in comparison to the hold-out method. The best model of these partitions is used as a final model which is less sensitive to the data partitions.

3.1. Unit Testing

Unit testing is one of the most popular regressions testing that can be implemented in Java code using JUnit[3] testing framework. It is a good practice to separate the testing code from the actual application code. EclEmma[4] is a free Java code coverage tool which is available under the Eclipse[5] public license; it is integrated with JUnit to execute test cases for Java program. It applies code coverage analysis into the

[3]http://junit.org/junit4

[4]https://marketplace.eclipse.org/content/eclemma-java-code-coverage

[5]https://eclipse.org

Eclipse workbench. JaCoCo[6] has been created by the EclEmma. It is a free Java code coverage library that runs JUnit test cases and provides a visual coverage report by providing information for each single test case that associated with each particular software module. As in the study [34], JaCoCo is the most popular testing model that gives higher visibility and easier integration for testing Java modules. The coverage information is most desirable when applying test cases optimization during regression testing because it highlights the covered code portions through running the test cases.

3.2. Imbalance Data and Oversampling

Software quality aims to develop high quality software by implementing machine learning classifiers on software metrics in order to discover the possible software defects modules. Machine learning [35] is frequently encountered with two main bottlenecks: working with imbalanced data and selecting the best features for machine learning techniques. Class imbalance [36]–[37] is a very common issue that occurs among several application domains such in software quality estimation. To overcome the difficulties associated with learning from imbalanced data, various techniques have been developed such as sampling. Data sampling is the primary technique for handling imbalanced distributions of software modules on a given dataset. According to this particular context, there are two types of data sampling: under-sampling of majority software modules and oversampling the minority ones as in [38]–[39]. In respect of the proposed approach, minority modules are buggy classes and majority are the buggy free ones. As a result, it is very essential to oversample the minority classes to make a better balancing with majority software modules.

3.3. Metrics and Tools

3.3.1. Static product metrics

Code metrics, also known as product metrics, measure the complexity of source code. This study uses code metrics to build several defects prediction models. All metrics are directly collected from an existing Java source code and calculated using Ckjm tool. The main reason behind using these metrics is to reflect good software quality attributes [1], [23], [40]. The static product metrics are selected from several metrics suites as follows:

A. The metrics suite are suggested by C&K:

1. Weighted method per class (WMC): Represents sum of the cyclomatic complexities of all declared local methods within a class.
2. Depth of inheritance tree (DIT): Measures class level in the inheritance tree, root class is considered as zero.
3. Number of children (NOC): Represents to the number of immediate children's or sub classes of a given class in the hierarchy.
4. Coupling between object classes (CBO): Measure the total number of new or redefined methods to which all the inherited methods are coupled.
5. Response for a Class (RFC): Measures the total number of local methods plus the number of non-local methods called by local methods.
6. Lack of cohesion in methods (LCOM): Measures the dissimilarity of methods in a particular class that shared at minimum one particular field or attribute. Lack of cohesion increases the likelihood of getting more defects.

B. Henderson-Sellers metric:

7. Lack of cohesion in methods (LCOM3): It is an extension from LCOM metric; it represents the connected components in the graph.

6http://eclemma.org/jacoco/

C. QMOOD metrics suite:
8. Number of Public Methods (NPM): Measures total number of all the public methods in a given class.
9. Data Access Metric (DAM): Represents proportion of the number of private or protected attributes to the overall number of declared attributes within a class.
10. Measure of Aggregation (MOA): Measures the total number of aggregation relationship between the class attributes.

11. Measure of Functional Abstraction (MFA): Represents percentage of number of inherited methods of an investigated class to the total number of methods that accessed by member methods of the class.
12. Cohesion Among Methods of Class (CAM): Computes the relatedness among methods of a class based on the parameters or specifications list of the methods.

D. Tang metrics:
13. Inheritance Coupling (IC): Indicates the number of parent or super classes that is coupled to a particular investigated class.
14. Coupling Between Methods (CBM): It represents the number of classes to which the investigated class is coupled.
15. Average Method Complexity (AMC). This metric measures the average method size for each class such as Java byte code length in the method.

E. Martin coupling metrics:
16. Afferent couplings (CA): Counts the number of classes that calling the investigated class.
17. Efferent couplings (CE): Counts the number of classes that called by the investigated class.

F. One McCabe's metric which represents class level complexity metrics:
18. McCabe's cyclomatic complexity (CC): Total number of all potential various paths for a given method within a class. It represents method's control flow and it defined as in (1).

$$\mathrm{CC} = \mathrm{E} - \mathrm{N} + \mathrm{P} \tag{1}$$

where E represents the number of edges of a graph, N represents the number of nodes of a graph, and P denotes the number of connected components. In addition, both of Max (CC) and Avg (CC) metrics have been derived from CC and stated as follows:

19. Max (CC): the maximum value of CC among all methods within a particular class.
20. Avg (CC): the arithmetic mean of all CC methods within a particular class.

G. Lines of Code
21. Line of code (LOC): Refers to the number of code lines for each class except the comments. Generally, a class with higher LOC number tends to contain higher defects than other classes.

3.3.2. Code Coverage Metrics

JUnit is one of the most popular regressions testing framework used by developers to execute unit testing for java source code. Our approach uses JaCoCo model which generates five code coverage[7] metrics named as: instruction, branch, line, complexity, and method. All these metrics are calculated by JaCoCo, the description of these metrics captured as follows:

A. Instructions coverage: Counts instructions into a single Java byte code. Instruction coverage represents the amount of code that has been executed by implementing JUnit test.

[7] http://www.eclemma.org/jacoco/trunk/doc/counters.html

B. Branches coverage: Counts branch coverage for all if and switch statements. This metric counts the total number of executed branches in a method within each class through implementing JUnit test.
C. Cyclomatic Complexity coverage: Measures all possible control flows within each concrete method within each class. It represents the number of test cases that required to fully covering the class code during the running of a particular JUnit test.
D. Lines coverage: For all class files that have been compiled with debug information, coverage information for individual lines can be calculated. A source line is considered executed when at least one instruction that is assigned to this line has been executed. It reflects the amount of cod e that has been exercised based on the number of Java byte code instructions called by the JUnit test.
E. Methods coverage: Each concrete method contains at least one instruction. A method is considered as executed when at least one instruction has been executed by implementing JUnit test.

All the above metrics are derived from Java byte code instructions and debug information which is existed in classes' files. We calculate the coverage ratio for each of code coverage metric by using (2).

$$\text{Coverage Ratio} = Covered\ Code/(Covered\ Code + Missed\ Code) \tag{2}$$

Where $Covered\ Code$- represents the amount of code that covered by running a given JUnit test case, and $Missed\ Code$- represents the amount of code that uncovered by executing a given JUnit test case. In order to find the most powerful metrics that correlated with the software bugs, , the proposed approach applies Pearson correlation on the all 21 static product and five code coverage metrics. For static product metrics, the recommended metrics are: WMC, CC, RFC, NPM, CBO, LCOM, CA, and CE correspondingly. On the other side, the highest correlated code coverage metrics are: Branch coverage, Instruction coverage, Method coverage respectively.

3.3.3. Bugs labeling

A bug, also known as defect, may lead to a total failure of software and eventually decrease the overall software quality. Therefore, this study identifies the bugs for each single class in each release of Apache Lucene project. Our proposed approach uses developers commenting guidelines that were stored in the source code repository (i.e. GitHub). As in the research study [25], a class is determined to be buggy if it contains at least one defect. So, our methodology pulls all the commits and their related comments for each release and consequently counts bugs for each class based on historical information that archived in software repository. GitHub[8] obviously includes information about all developers' commits description. During the interim development, our approach counts the number of defects for a given class within each current release by considering the number of repaired defects within the same class of the previous release.

3.4. The Proposed Approach

In the context of our study, four essential datasets are created as shown in Fig. 1. As a result, four essential scenarios are used with the purpose of building software defects prediction models:

(1) Static product scenario (Scenario 1). This scenario mainly concentrates on using only static product metrics with the associated bugs to produce static product prediction model.
(2) Code coverage scenario (Scenario 2). This scenario essentially focuses on using only code coverage metrics with the associated bugs to get code coverage prediction model.
(3) Hybrid scenario (Scenario 3). This scenario combines both of scenario 1 and scenario 2 metrics with their corresponding bugs to create hybrid prediction model.

[8]https://developer.github.com/v3/repos/commits/

(4) Feature selection scenario (Scenario 4). This scenario considers the key features of the hybrid scenario with the related bugs to obtain feature selection prediction model.

The 26 independent variables (input) for both static product and code coverage metrics, the only dependent variable is the bug (output). The purpose of defect prediction is to determine defective classes in each software release.

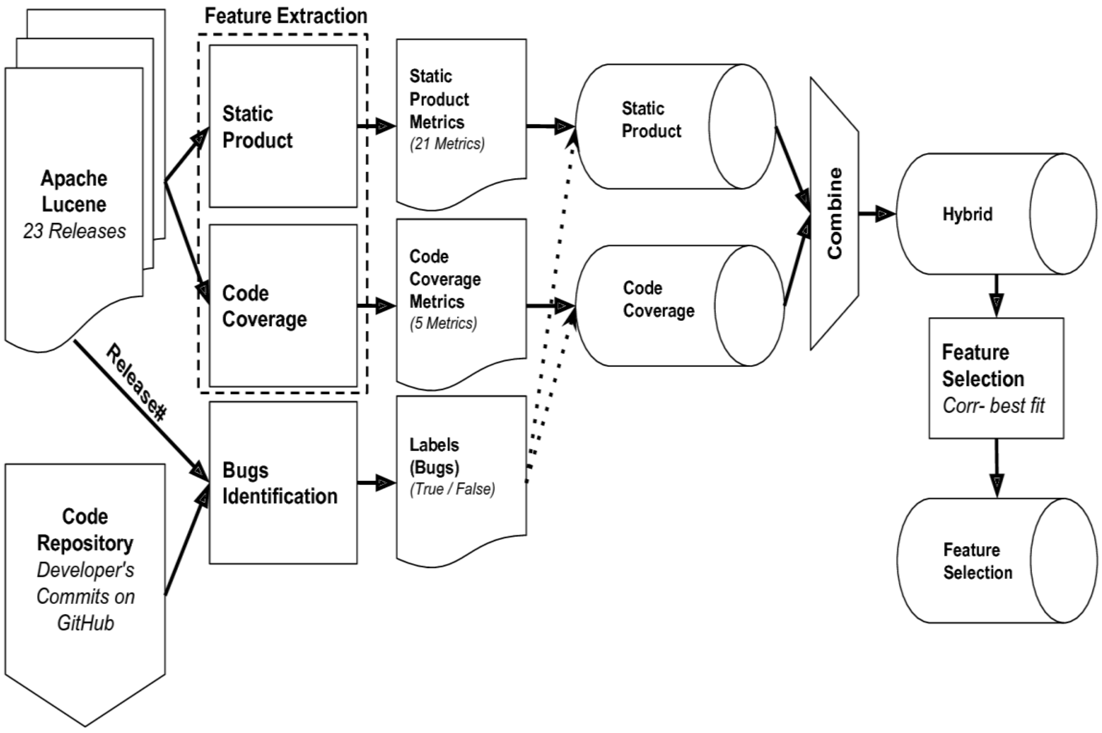

Feature Extraction
Apache Lucene 23 Releases
Static Product
Code Coverage
Static Product Metrics (21 Metrics)
Code Coverage Metrics (5 Metrics)
Static Product
Code Coverage
Combine
Hybrid
Feature Selection Corr- best fit
Feature Selection
Release#
Code Repository Developer's Commits on GitHub
Bugs Identification
Labels (Bugs) (True / False)

Fig. 1. Dataset creation.

Our proposed methodology consists of seven ordered phases, it combines the strength of static and dynamic aspects of the source code, static aspect focuses on static product metrics while dynamic aspect highlights the code coverage metrics, Fig. 2 summarizes the steps:

Phase1: Collect Java open source code. Our approach collects 23 successive releases from Apache Lucene project, Lucene is the standard Java libraries [41] that used for different purposes such as searching, indexing, and tokenizing.

Phase2: Measure static product metrics for each module. All 21 static product metrics are directly collected from an existing source code by using Ckjm. One of the most important key features of this tool is to report the key quality indicators in the body of Java source code.

Phase3: Measure software code coverage metrics for each module by using JaCoCo.

Phase4: Classify software modules into defective and non-defective by using an attribute called bug count, the value of bug count attribute is assigned to true (i.e. if the module contains bug) or false otherwise. Software tester should begin the software testing with a software module that contains a high number of defects in order to avoid extra cost. Our approach determines if the module is defective or not by analyzing the historical information that archived in GitHub.

Phase5: Combine all the dataset (static product, code coverage, hybrid, and features selection) by applying 10 folds cross validation over all 23 investigated releases. The proposed approach uses 13 various machine learning classifiers.

Phase6: Oversample all the datasets to obtain more balanced distribution between defective and non-defective software modules.

Phase7: Evaluate the proposed methodology for two rounds of experiments by using recall, precision, F-measure, and ROC.

4. Results and Discussions

4.1. Dataset Description

In our study, 23 successive releases of Apache Lucene project have been investigated in order to evaluate the proposed software defects prediction models. Apache Lucene is one of the most popular Java open source projects as well as it is accompanied with JUnit test cases which can be executed via JaCoCo. Table 1 presents the statistical information about our collected datasets, where the last column is the ratio of buggy modules for each release. Each module in this dataset represents a class file in a Lucene release and it contains 26 metrics (independent variables) and bug (only one dependent variable).

The bug value is converted into binary labels; a module is non-buggy if it has zero bugs. Otherwise, it is buggy. All static product and code coverage metrics that are involved in our approach are described previously in section 3.3.

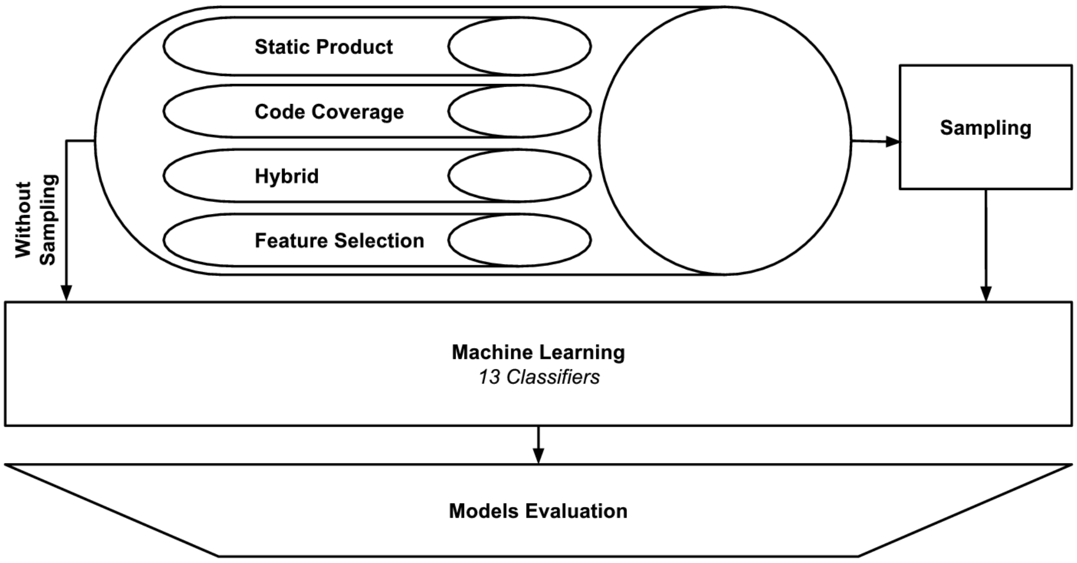


Fig. 2. The proposed approach.

4.2. Experimental Setup

Our experiments have two rounds, first round considers without sampling and second round applies oversampling of the buggy software modules. Each experiment applies 13 machine learning classifiers on the 23 investigated Apache Lucene releases. These classifiers belong to five different classification families and named as: AdaBoostM1, Bagging, IBK, J48, KStar, LWL, Multi Layers Perceptron (MLP), Naive Bays (NB), Random Forest, SGD, Simple Logistic, SMO, and logitBoost.

4.3. Performance Measures

In this approach, binary classification approach is used to predict classes that are likely to contain defects. A binary classifier can make two potential errors: False Positive (FP) and False Negative (FN). Also, a properly classified buggy class is a True Positive (TP) and properly classified non-buggy class is a True Negative (TN). We evaluated binary classification results in terms of recall, precision, F-measure and ROC.

Table 1. Dataset Statistics

Apache Lucene Release	No.of defect modules	No. of all modules	Defective ratio
5.2.0	278	3079	9.0%
5.2.1	278	3082	9.0%
5.3.0	277	3192	8.6%
5.3.1	277	3193	8.6%
5.3.2	277	3193	8.6%
5.4.0	299	3255	9.1%
5.4.1	300	3255	9.2%
5.5.0	317	3282	9.6%
5.5.1	317	3282	9.6%
5.5.2	319	3282	9.7%
5.5.3	319	3282	9.7%
5.5.4	319	3282	9.7%
6.0.0	425	3113	13.6%
6.0.1	425	3113	13.6%
6.1.0	438	3162	13.8%
6.2.0	448	3187	14.0%
6.2.1	448	3194	14.0%
6.3.0	450	3232	13.9%
6.4.0	445	3297	13.4%
6.4.1	445	3297	13.4%
6.4.2	447	3297	13.5%
6.5.0	454	3339	13.5%
6.5.1	455	3339	13.6%

Recall: calculates the number of defective classes that are actually estimated by a prediction model. A higher recall indicates a lower number of defective classes that have been missed by the prediction model as in (3).

$$\text{Recall} = \text{TP}/(\text{TP} + \text{FN}) \tag{3}$$

Precision: calculates the number of defective classes that are estimated by a prediction model. The higher precision indicates a lower non-defective class on instances as in (4).

$$\text{Precision} = \text{TP}/(\text{TP} + \text{FP}) \tag{4}$$

F-measure: calculates the accuracy of prediction model based on both precision and recall, which can be interpreted as a weighted average of both precision and recall. The higher the F-measure indicates better performance for classification results as in (5).

$$\text{F} - \text{measure} = (2 * \text{Precision} * \text{Recall})/(\text{Precision} + \text{Recall}) \tag{5}$$

Receiver Operating Characteristic (ROC): is used typically to calculate the performance of prediction models by comparing the true positive metric to false positive metric on a threshold value (depends on number of instances). The area under curve of two metrics is considered as a value of evaluation.

4.4. Experimental Results

The results of both investigated experiments are reported in order to answer the four stated research questions above. The detailed description of both experiments is shown as in the following subsections 4.4.1, 4.4.2 respectively.

4.4.1. Without sampling

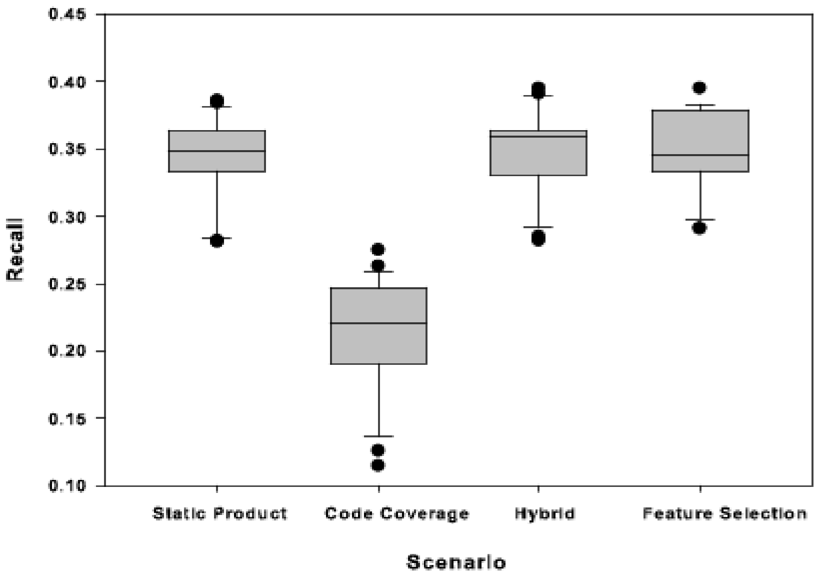
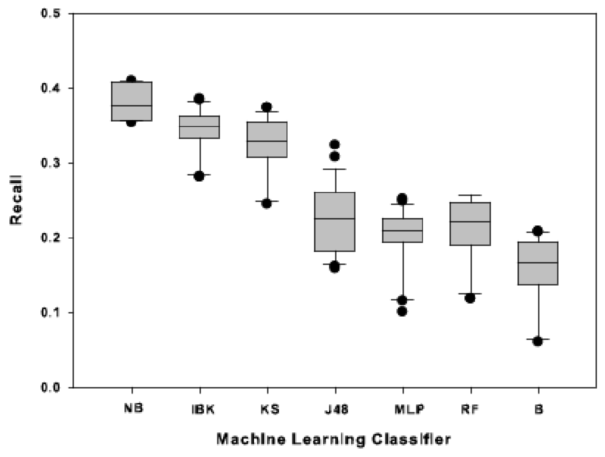


Fig. 3. Performance of NB classifier on all scenarios in terms of recall.

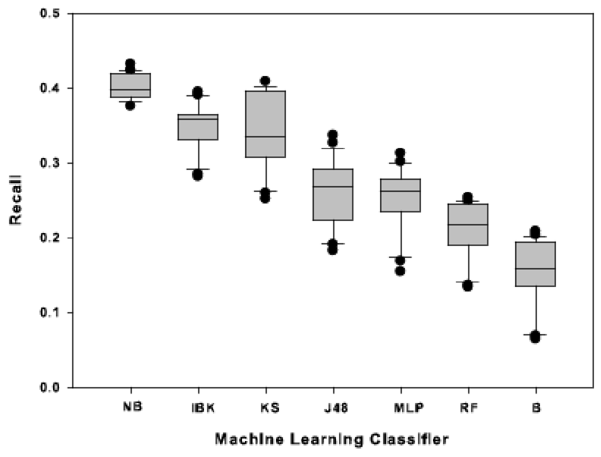
We build our research questions based on adapting different scenarios that potentially reflect the effect of combining code coverage metrics with the previous static product metrics in order to predict the software defects. Fig. 3 shows box-plots of the recall values over all 23 releases of the NB classifier by investigating the four stated scenarios. NB is marked as the best machine learning classifiers based on the achieved recall values. The feature selection scenario superiors others scenarios where approximately 50% of recall values are more than 35% of a graph skewed to the left. The hybrid scenario has comparable performance to static product scenario where the median recall values for both are between 28% and 33%. While the code coverage scenario shows the worst results due to the few metrics as input for the learning classifier and the executed test cases achieve partial or even no coverage, low code coverage ratio reflects that the code is not tested and classified as buggy code while high coverage modules reflect higher quality of source code. However, the code coverage metrics have an improvement effect when they are used as features in hybrid scenario.

The results of the first experiment raise RQ1 where the code coverage metrics have significant improvement in software defect prediction especially when there is high coverage ratio. Our further discussions have to focus on other three scenarios: static product, hybrid, and feature selection. To address RQ2, the three suggested scenarios for software prediction are investigated by analyzing the performance of classifiers using the picked dataset without sampling. Fig. 4(a), 4(b), and 4(c) illustrate prediction algorithms performance for the static, hybrid, and feature selection scenarios respectively.

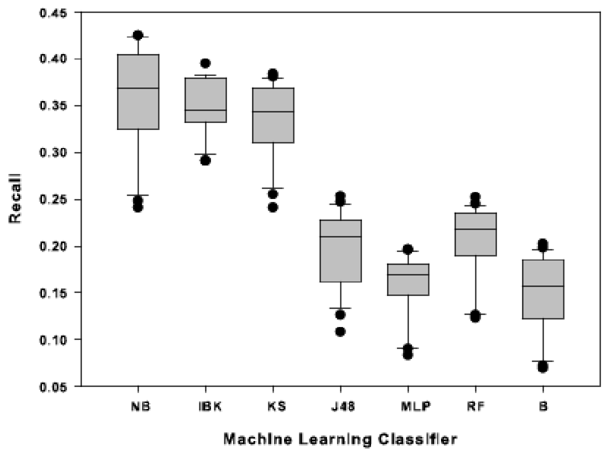
The static product scenario as shown in Fig. 4(a) nominates NB, IBK, and KS as a top three performing classifiers, the recall mean values are 0.38, 0.34, and0.32respectively. The hybrid scenario is presented in Fig. 4(b) nominates NB, IBK, and KS as top three performing classifiers as in the static product scenario. The exact mean values are 0.40, 0.35, and 0.34 respectively. Similarly, feature selection as shown in Fig. 4(c) nominates the top three classifiers: NB, IBK, and KS with recall mean of 0.35, 0.35, and 0.33 respectively. Adding code coverage metrics to static product metrics leveraged the prediction accuracy and improves overall software quality as illustrated in the hybrid scenario.



(a)



(b)



(c)

Fig. 4. Top seven classifiers in terms of prediction performance: (a) Static product scenario, (b) Hybrid scenario and (c) Feature selection scenario.

Such improvement in software quality leads to assert that using of code coverage metrics is important to obtain better prediction of software defects. It is apparent that NB, IBK, and KS surpass significantly the rest of all classifiers. In essence, NB is considered as the most suitable classifier in the software defect prediction models over the three selected scenarios. The suitability of using NB for software defect prediction model agrees with the characteristics of the algorithm itself, it can be better than several classifiers under default configurations as well as it is appropriated for binary classification problems as concluded by the study [42]. The overall analysis of the results provides the following answers: (1) in response to RQ1, the hybrid scenario improves software defect prediction since it considers code coverage metrics, (2) in terms of RQ2; all scenarios agreed that the most appropriate classifiers are NB, IBK, and KS.

In fact, according to the recall, the top seven classifiers are: (1) NB, IBK, KS, MLP, J48, RF, and B for hybrid scenario; (2) NB, IBK, KS, RF, MLP, J48, and B for static product metrics scenario; and (3) NB, IBK, KS, RF, J48, MLP, and B for feature selection scenario. Among all classifiers, NB was the most efficient classifier that used in software defect prediction.

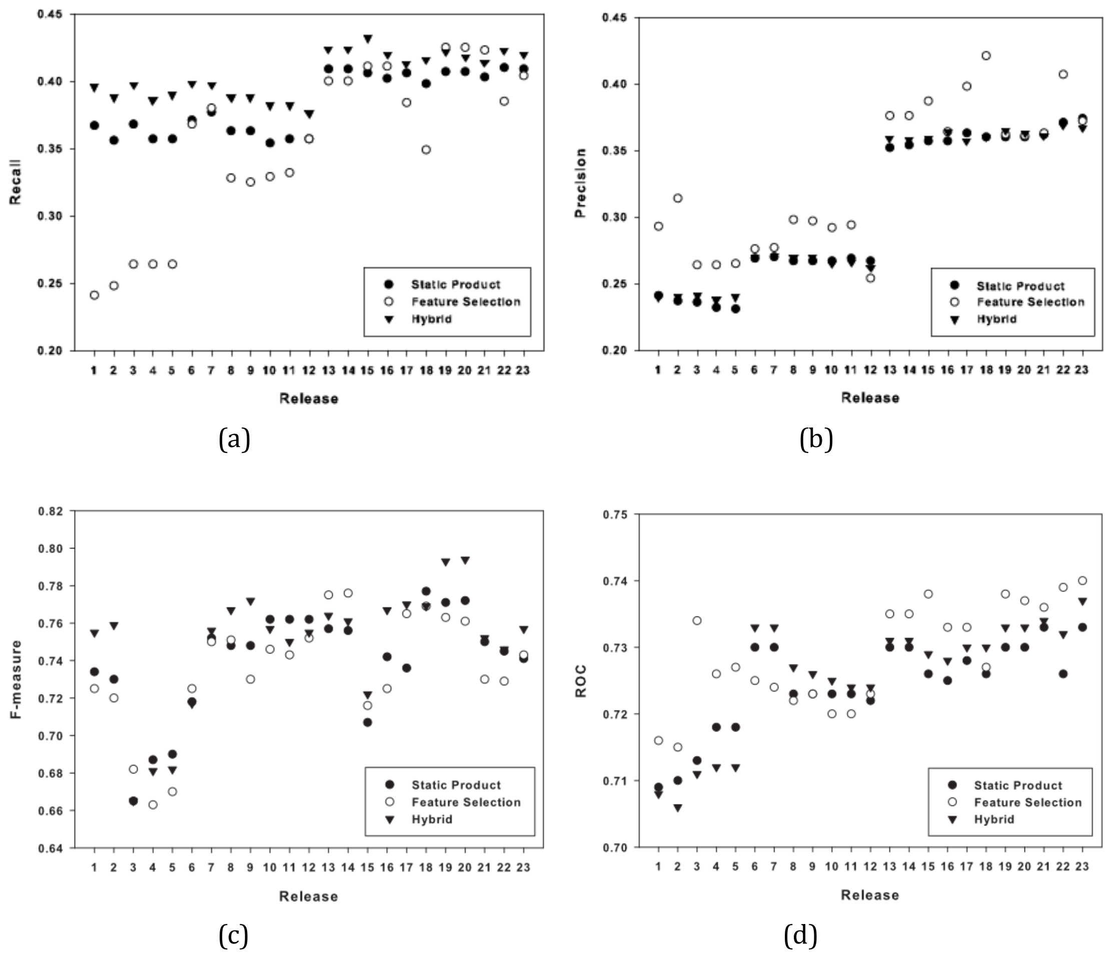
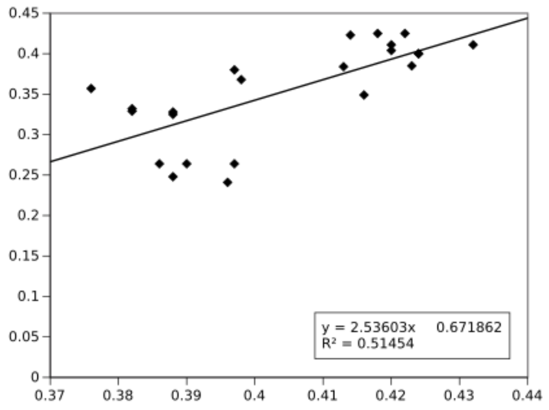


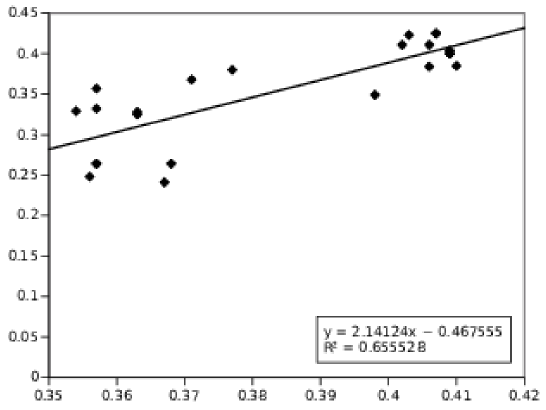
Fig. 5. The behavior of all 23 Lucene releases under NB classifier in respect to the measures: (a) Recall, (b) Precision, (c) F-measure, and (d) ROC.

In general, most of the later releases show a steady improvement with comparing to the prior releases in terms of all four evaluation measures as shown in Fig. 5 (a), Fig. 5 (b), Fig. 5 (c), and Fig. 5 (d) respectively. According to our perspective, recall is considered to be the most key measure among all used measures because our goal is to predict False Negative (i.e. non-buggy classes that predicted as buggy). In particular to NB classifier recall of hybrid scenario, there is a steady significant improvement except from release 6 (Apache Lucene 5.4.0) to release 12 (Apache Lucene 5.5.4). This occurs due to many reasons such: very few classes that have been added to these particular releases, modification lines of code within certain classes, and possibility of low coverage ratio existence. On other side, there is new remarkable milestones between release 12 (Apache Lucene 5.5.4) and release 13 (Apache Lucene 6.0.0) as well as between release 5 (Apache Lucene 5.3.2) and release 6 (Apache Lucene 5.4.0) that happens for significant updates between these exact releases. Our proposed approach uses 10-fold cross validation technique. Through all investigated Lucene releases, hybrid outperforms the performance of the other scenarios because it has the best recall as shown in Fig. 5 (a).

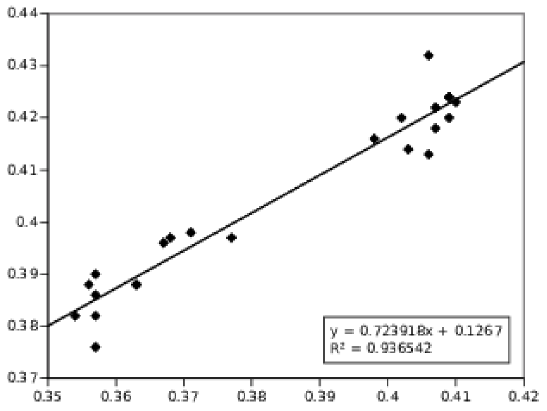
To capture the releases behavior under different measures as stated in RQ4, three different releases are explored: release 2 (Apache Lucene 5.2.1), release 17 (Apache Lucene 6.2.1), and release 22 (Apache Lucene 6.5.0). First, the recall values of hybrid scenario are 38%, 41%, 43% for release 2, release 17, and release 22 respectively. Second, the F-measure values of hybrid scenario are 29%, 38%, 39% for release 2, release 17, and release 22 respectively. In general, there is monotonic increasing for software releases in all evaluation measures.



(a)



(b)



(c)

Fig. 6. Correlation coefficient between two different scenarios with no sampling experiment: (a) Hybrid vs. Feature selection scenarios; (b) Static vs. Feature Selection scenarios; and, (c) Static vs. Hybrid scenarios.

Also, based on the recall values of NB classifier, it is very important to study the correlations between the discussed scenarios and see their strength in predicting of the software defects. As a result, in order to compare the quality of predicting defects by picking certain machine learning classifiers and evaluation measure, this study examines the relationship among our scenarios to discover the behavior of releases by moving from one release to another. Fig. 6 shows a scattered plot which represents the correlation relationship among different scenarios in without-sampling experiment. Each plot contains a linear regression equation that relates the recall values of the first scenario with the second scenario. In addition, it clarifies the R^2 correlation coefficient. The R^2 is a statistical measure of how close the real data points are fitted by the linear regression model. This means that if the R^2 value is close to 1, the data is highly fitted to the regression line and there is no difference in their effects on software defects prediction and has the same impacts. In particular, as shown in Fig. 6, the static and hybrid scenarios have high R^2 values which approximately 0.94 and a line slop value 0.724 where the data is almost on the regression line.

This indicates that the metrics used in the hybrid scenario are close in term of effectiveness to the static product metrics and also ensures the importance of using the code coverage metrics. On the other hand, feature selection scenario has no relation with other two scenarios; hybrid in (a) R^2 value 0.51454 and static in (b) R^2 value 0.655528. In addition to that, on each scatters plot the recall values are monotonically increased (positive slope) from earlier releases to the later ones.

4.4.2. With sampling

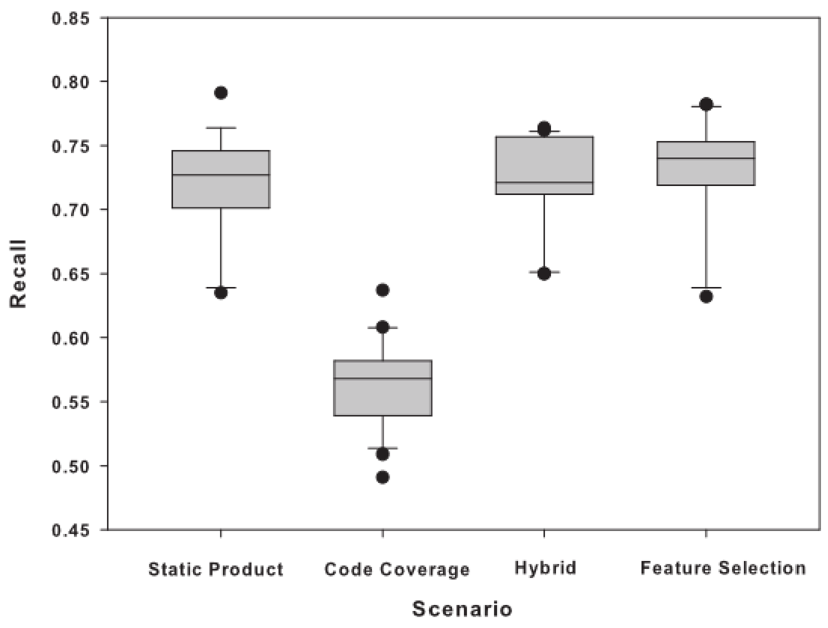
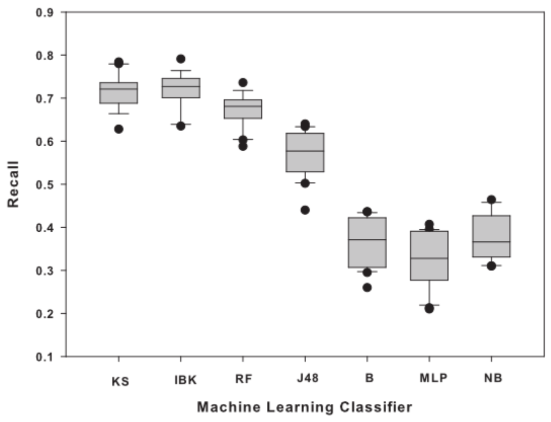


Fig. 7. Performance of IBK classifier on all scenarios in term of recall.

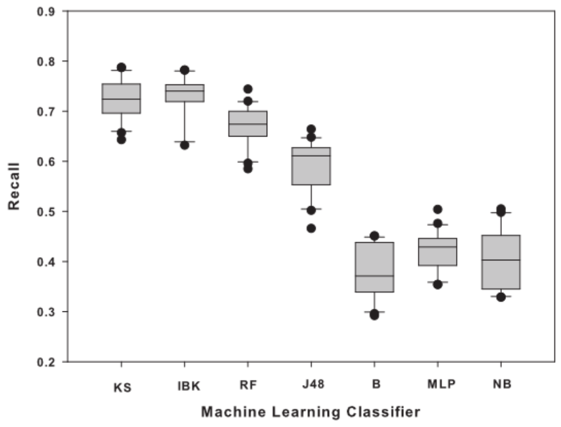
Due to imbalanced dataset, since the buggy classes are the minor classes, the proposed approach applies oversampling on the four primary datasets, Fig. 7 shows box-plots of recall values of the IBK classifier using the four proposed scenarios. The IBK classifier achieved the highest recall value among all 13 classifiers. The results concluded that feature selection and hybrid scenarios outperform the static product and cod e coverage scenarios, the mean recall values are 73% and 72% correspondingly. In respect to RQ1, the code coverage scenario in this experiment shows two time's better recall rather than the first experiment. Such improvements refer to increase the buggy classes that manipulated through the oversampling process. Obviously, the code coverage metrics improve the performance of both hybrid and feature selection scenarios. In regards with RQ1, the results raise that using of code coverage metrics mirror remarkable improvement in software defect prediction. Undoubtedly, the oversampling methodology improves the prediction of software defects in comparison with the first round of experiment, which obviously defends the issue of RQ3. Fig. 8 illustrates the performance of the top seven classifiers in terms of recall. In particular, Fig. 8(a), Fig. 8(b), and Fig. 8(c) illustrate the classifiers performance for static product, hybrid, and feature selection scenarios respectively.

Across the three scenarios, IBK has a highest accuracy with comparing to the other classifiers. Fig.8 (a), 8(b), and 8(c) show that the recall average values of IBK are 71.98%, 72.91%, and 72.22% respectively. Also, in comparison with the first experiment, all the three scenarios have significant increase in recall value. IBK shows the best performance in this experiment, it outperforms other classifiers due to: (1) more balanced training, (2) it relatively requires less time to classify modules as in [43], and (3) in particular, it almost shows a good classification performance for numeric and nominal datasets [44].

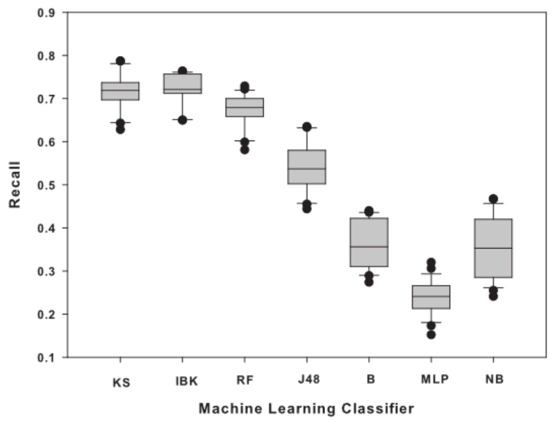
On the other hand, standard NB behaves differently in this experiment as concluded in the study [45]. But it performs better if PCA-based preprocessing is applied to the training data, or re-weighted NB is used. At the end, the experimental results show that NB is the most appropriate classifier in first experiment, while IBK is the most suitable one in the second experiment, this conclusion clearly answers RQ2. Moreover, in response to RQ1, using code coverage metrics strengthen software defect prediction as it appears in hybrid and feature selection scenarios.



(a)

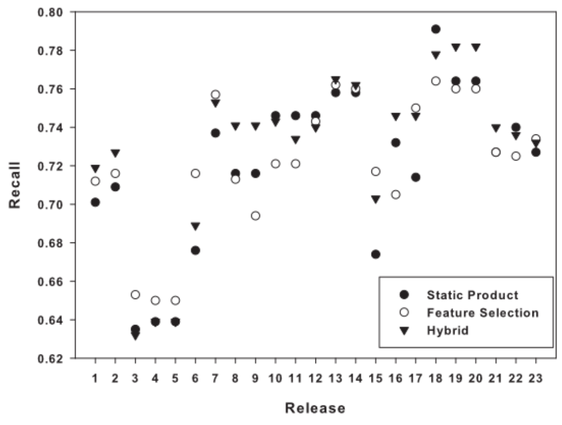


(b)

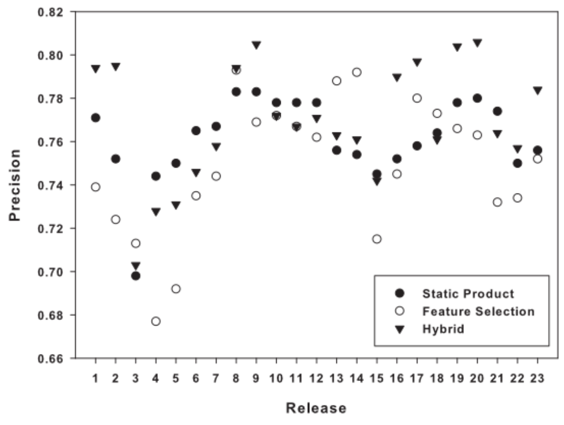


(c)

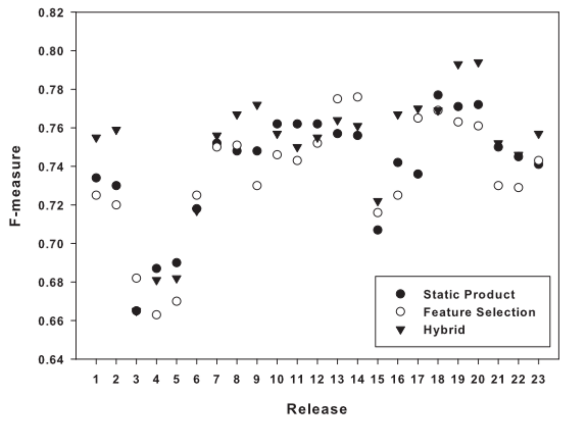
Fig. 8. Top seven performing classifiers with sampling in term of Recall measure: (a) Static product scenario; (b) Hybrid scenario; and, (c) Feature selection scenario.



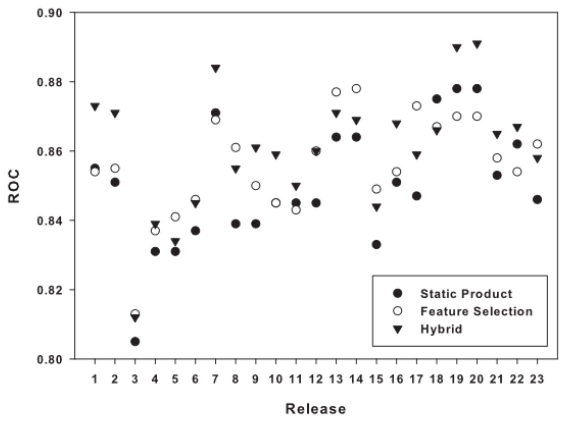
(a)



(b)



(c)



(d)

Fig. 9. Behavior of all 23 Lucene releases under IBK classifier with sampling in terms of: (a) Recall, (b) Precision, (c) F-measure, and (d) ROC.

In contrast to the first experiment, oversampling experiment has skewed behavior through releases because there is no noticeable harmony as shown in Fig. 9(a), Fig. 9(b), Fig. 9(c), and Fig. 9(d). However, hybrid and feature selection scenarios superior the other scenarios such interesting findings happen due to the fact that oversampling methodology handles random flavor of normal distribution of the datasets. As a result, we cannot predict the class replacements that occur through the sampling. For example, we cannot track the replacement nature of a certain class through all software releases.

In conclusion, there is a random behavior of Lucene releases in respect to the four used evaluation measures. First, the recall values of feature selection scenario are 71.9%, 74.9%, 72% for release 2, release 17, and release 22 respectively as in Fig. 9(a). Second, the precision values of feature selection scenario are 72.4%, 78%, 73% for release 2, release 17, and release 22 subsequently as in Fig. 9(b). Third, the F-measure values of feature selection scenario are 72%, 76.4%, 73% for release 2, release 17, and release 22 consequently as in Fig. 9(c). Fourth, the ROC values of feature selection scenario are 85.5%, 87%, 85% with respect to release 2, release 17, and release 22 as in Fig. 9(d). The results answer RQ4 by expressing the detection behavior for the releases in the second round of experiments. The sampling techniques, no doubt, tackle the problems of imbalance data inherited in training dataset to consolidate learning methods in estimating the defective classes. The question comes up here; is there any effect of using sampling method on the correlations with the scenarios in the releases.

This study clearly shows the relationship among scenarios to investigate the behavior of releases as discussed earlier in the first experiment. Fig. 10 shows a scattered plot that presents the correlation among the three scenarios in oversampling experiment.

Notably, as shown in Fig. 10, the static product and hybrid scenarios have the highest R^2 values of approximately 0.91 and a line slop of 0.96 where the data is almost on the regression line as in first round, this denotes that the features used in the hybrid scenario are also close in the effectiveness to the static product metrics and asserts the value of using code coverage metrics within hybrid scenario. Besides, the feature selection scenario has no relation with other two scenarios; Hybrid scenario in (a) of R^2 value is approximately 0.82 and static product scenario in (b) of R^2 value is 0.77. Furthermore, the results show that the slopes seem to be ideal to ensure the closeness of relations between scenarios with little differences between Hybrid vs. Feature selection and Static product vs. Feature Selection scenarios. As comparison with first round, the results clearly show that the Hybrid vs. Feature selection correlation has better performance rather than in the first round.

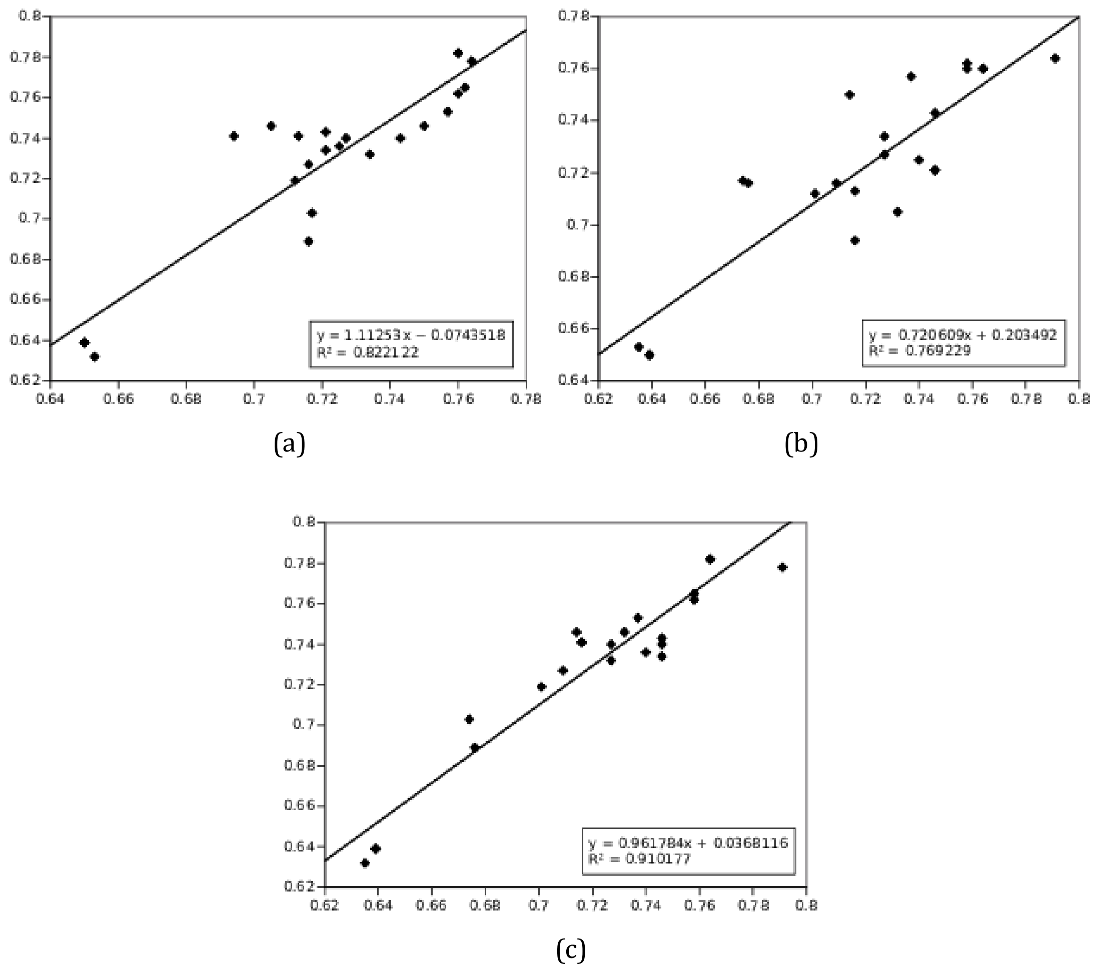


Fig. 10. Correlation coefficient between two different scenarios with sampling: (a) Hybrid vs. Feature selection scenarios; (b) Static vs. Feature Selection scenarios; and, (c) Static vs. Hybrid scenarios.

Several studies [46]–[48] investigate Lucene in order to predict software defects. These studies, however, do not tackle the code coverage features. Instead, they consider only baseline metrics. This paper explores five code coverage metrics over 23 consecutives releases of Lucene in order to emphasize the coverage role of the source code in software defect prediction. With regarding to the first experiment, the recall values for the approaches in [46]–[47] are 37.8%, 31.5% respectively, while the recall of the proposed approach is 43%. Also, in respect to the second experiment, the recall of their approach in the study [48] is 64.1%, whereas the recall of the proposed approach is 71.9%. Our results show a noticeable improvement of software defect prediction due to the use of code coverage metrics through considering JUnit testing for all investigated software modules.

This proposed approach differs from the previous approaches [23]–[29] in the literature because it adds code coverage metrics to the past used metrics. The results show the significant role of combining code coverage metrics with static product metrics will improve the accuracy of software defect predication. Our experiments shows that using sampling can attain better prediction because it allows machine learning classifiers to obtain rich information that increase the prediction accuracy as suggested in [49]–[50].

5. Conclusion and Perspectives

This study explored the possibilities to improve software defect prediction. Better prediction of software defects improves overall software quality. Additionally, it facilitates an efficient resource allocation for software testing. In order to achieve this goal, this research proposed a novel approach that leverages capabilities of code coverage metrics to consolidate the conventional static characteristics of source code. The quality of code coverage enables quantifying the amount of changes that happened to the source code.

In order to validate the proposed approach, four distinct scenarios for software defect prediction have been employed. First scenario maintains static product metrics, second scenario maintains metrics conveying code coverage results while the other two scenarios are built based on aggregation of the metrics obtained by the first two scenarios. Results indicate a significant improvement in the prediction rates due to the consideration of code coverage. Further, it exhibits the superiority of both NB and IBK as machine learning classifiers among thirteen distinct classifiers investigated. Due to imbalanced, an oversampling technique is applied on minority software modules, it significantly improves the prediction of software defects by obtaining two times better recall comparing with the first round of experiment. Finally, across all four used evaluation measures, there is a steady improvement from earlier releases to later ones and the promising results from our evaluation show that our approach can be applied into practice. As a future work, we aim to apply the proposed approach to industrial projects.

6. References

[1] Carlson, R., Do, H., & Denton, A. (2011). A clustering approach to improving test case prioritization: An industrial case study. *Proceedings of the 27th IEEE International Conference on Software Maintenance* (pp. 382 – 391).

[2] He, P., Li, B., Liu, X., Chen, J., & Ma, Y. (2015). An empirical study on software defect prediction with a simplified metric set. *Information and Software Technology.*

[3] Hewett, R. (2011). Mining software defect data to support software testing management. *Applied Intelligence, 34(2),* 245–257.

[4] Jureczko, M. (2011). Significance of different software metrics in defect prediction. *Software Engineering: An International Journal.*

[5] Meneely, A., Williams, L., Snipes, W., & Osborne, J. (2008). Predicting failures with developer networks and social network analysis. *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (pp. 13 – 23).

[6] Stringfellow, C., & Andrews, A A. (2002b). An empirical method for selecting software reliability growth models. *Empirical Software Engineering, 7(4).*

[7] Menzies, T., Greenwald, J., & Frank, A. (2007). Data mining static code attributes to learn defect predictors. *IEEE Transactions on Software Engineering.*

[8] Nagappan, N., & Ball, T. (2007). Using software dependencies and churn metrics to predict field failures: An empirical case study. *Empirical Software Engineering and Measurement.*

[9] Ostrand,T. J., Weyuker, E. J., & Bell, R. M. (2005). Predicting the location and number of faults in large software systems. *IEEE Transactions on Software Engineering.*

[10] Turhan, B., & Bener, A. (2007). A multivariate analysis of static code attributes for defect prediction. *Seventh International Conference on IEEE.*

[11] Caglayan, B., Misirli, A. T., Bener, A. B., Miranskyy, A. (2015). Predicting defective modules in different test phases.*Software Quality Journal.*

[12] Hewett, R. (2011).Mining software defect data to support software testing management. *Applied Intelligence, 34(2),* 245–257.

[13] Andrews, A., & Stringfellow, C. (2001). Quantitative analysis of development defects to guide testing: A case study. *Software Quality Journal, 9(3),* 195–214.

[14] Khoshgoftaar, T. M., Szabo, R. M., & Woodcock, T. G. (1994). An empirical study of program quality

during testing and maintenance. *Software Quality Journal, 3(3),* 137–151.

[15] Stringfellow, C., & Andrews, A. (2002a). Deriving a fault architecture to guide testing. *Software Quality Journal 10(4),* 299–330.

[16] Kastro, Y., & Bener, A. B. (2008).A defect prediction method for software versioning. *Software Quality Journal.*

[17] Chidamber, S. R., & Kemerer, C. F. (1994). Ametrics suite for object oriented design. *IEEE Transaction Software Engineering.*

[18] Henderson-Sellers, B. (1996).*Object-oriented Metrics: Measures of Complexity*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

[19] Martin, R. (1994). OO design quality metrics.An analysis of dependencies.

[20] Bansiya, J., & Davis, C. G. (2002). A hierarchical model for object-oriented design quality assessment. *IEEE Transactions on Software Engineering,.*

[21] Tang, M. H., Kao, M. H., & Chen, M. H. (1999). An empirical study on object-oriented metrics. *Proceedings of the Sixth International IEEE on Software Metrics Symposium* (pp. 242–249).

[22] McCabe, T. J. (1976). A complexity measure. *IEEE Transactions on software Engineering.*

[23] Jacob, S. G.,et al. (2015).Improved random forest algorithm for software defect prediction through data mining techniques. *International Journal of Computer Applications.*

[24] Jureczko, M. (2011). Significance of different software metrics in defect prediction. *Software* Engineering. *An International Journal.*

[25] Jureczko, M., & Madeyski, L. (2010). Towards identifying software project clusters with regard to defect prediction. *Proceedings of the 6th International Conference on Predictive Models in Software Engineering.*

[26] Shatnawi, R. (2012).Improving software fault-prediction for imbalanced data. International Conference on IEEE: Innovations in Information Technology (IIT), (pp. 54 – 59).

[27] Tang, H., Lan, T., Hao, D., & Zhang, L. (2015). Enhancing defect prediction with static defect analysis.*P roceedings of the 7th Asia-Pacific Symposium on Internetware.*

[28] Tomar, D., & Agarwal, S. (2016). Prediction of defective software modules using class imbalance learning. *Applied Computational Intelligence and Soft Computing.*

[29] Madeyski, L., & Jureczko, M. (2015). Which process metrics can significantly improve defect prediction models? An empirical study.*Software Quality Journal.*

[30] Witten, I. H., Frank, E., & Hall, M. A. (2011). *Data Mining: Practical Machine Learning Tools and Techniques*.

[31] Davis, J., & Goadrich, M. (2006). The relationship between precision-recall and roc curves. *Proceedings of the 23rd International Conference on Machine Learning.*

[32] Lusted, L. B. (1971). Signal delectability and medical decision-making. Science.

[33] Kohavi, R. (1995).The power of decision tables. *Proceedings of the European Conference on Machine Learning.*

[34] Horv´ath, F., Gergely, T., Besz´edes, A., Tengeri, D., Balogh, G., Gyim´othy, T. (2017). Code coverage differences of java byte code and source code instrumentation. *Software Quality Journal.*

[35] Khoshgoftaar, T M. Gao, K., & Seliya, N. (2010). Attribute selection and imbalanced data: Problems in software defect prediction. *Proceedings of the 22nd IEEE International Conference,Tools with Artificial Intelligence.*

[36] Jiang, Y., Lin, J., Cukic, B., & Menzies, T. (2009). Variance analysis in software fault prediction models. *Proceedings of the 20th International Symposium on Software Reliability Engineering.*

[37] Khoshgoftaar, T. M. Rebours, P., & Seliya, N. (2009). Software quality analysis by combining multiple

projects and learners. *Software quality journal, 17(1)*, 25–49.

[38] Chawla, N. V., Bowyer, K. W. Hall, L. O., & Kegelmeyer, W. P. (2002). Smote: Synthetic minority oversampling technique. *Journal of Artificial Intelligence Research.*

[39] Seiffert, C., Khoshgoftaar, T. M., Van, H., J. (2009). Improving software quality predictions with data sampling and boosting. *IEEE Transactions on Systems, 39(6),* 1283–1294.

[40] Paramshetti, P., & Phalk, D. (2015). Software defect prediction for quality improvement using hybrid approach. *International Journal of Application or Innovation in Engineering & Management.*

[41] Azzopardi, L., Moshfeghi, Y., Halvey, M., Alkhawaldeh, R. S., Balog, K., Di, B. E., Ceccarelli, D., Fern´andez-Luna, J. M., Hull, C., Mannix, J., & Palchowdhury, S. (2017). Lucene4ir: Developing information retrieval evaluation resources using lucene, *50(2),* 58–75.

[42] Amancio, D. R., Comin, C. H., Casanova, D., Travieso, G., Bruno, O. M., Rodrigues, F. A., Fontoura, C. L. (2014). A systematic comparison of supervised classifiers.

[43] Yıldırım, P. (2016). Pattern classification with imbalanced and multiclass data for the prediction of albendazoleadverse event outcomes. *Procedia Computer Science*, *83*, 1013–1018.

[44] Bostanci, B., & Bostanci, E. (2013). An evaluation of classification algorithms using mc nemars test. *Proceedings of Seventh International Conference on Bio-Inspired Computing: Theories and Applications.*

[45] Turhan, B., & Bener, A. (2009). Analysis of naive bayes assumptions on software fault data: An empirical study. *Data & Knowledge Engineering.*

[46] Wang, S., Liu, T., & Tan, L. (2016). Automatically learning semantic features for defect prediction. *Proceedings of the 38th International Conference on Software Engineering* (pp. 297–308).

[47] Tan, M., Tan, L., Dara, S., & Mayeux, C. (2015). Online defect prediction for imbalanced data. *Proceedings of the 37th International Conference on Software Engineering.*

[48] Li, M., Zhang, H., Wu, R., & Zhou, Z.-H. (2012).Sample-based software defect prediction with active and semi-supervised learning. *Automated Software Engineering Journal.*

[49] Hassan, E. (2009). Predicting faults using the complexity of code changes. *Proceedings of the 31th International Conference on Software Engineering, Canada).*

[50] Zimmermann, T., & Nagappan, N., (2008). Predicting defects using network analysis on dependency graphs. *Proceedings of the 30th International Conference on Software Engineering, Germany* (pp. 531–540).

Bilal Al-Ahmad received B.Sc. degree in computer information systems from Jordan University of Science & Technology, Jordan, in 2006, M.Sc degree in computer information systems from Yarmouk University, Jordan, in 2009, PhD in software engineering from North Dakota State University, USA, in 2015. Currently, he is an assistant professor in Computer Information Systems department at The University of Jordan, Aqaba branch. His research interests include requirements engineering, software testing, software design, and machine learning.