# Method Level Refactoring Prediction on Five Open Source Java Projects using Machine Learning Techniques

Lov Kumar
BITS Pilani - Hyderabad
Hyderbad, India
lovkumar505@gmail.com

Shashank Mouli Satapathy
Vellore Institute of Technology
Vellore, India
shashankamouli@gmail.com

Lalita Bhanu Murthy
BITS Pilani - Hyderabad
Hyderbad, India
bhanu@hyderabad.bits-pilani.ac.in

## ABSTRACT

Introduction : Identifying code segments in large and complex systems in need of refactoring is non-trivial for software developers. Our research aim is to develop recommendation systems for suggesting methods which require refactoring. *Materials and Methods :* Previous research shows that source code metrics for object-oriented software systems are indicators of complexity of a software system. We compute 25 different source code metrics at the method level and use it as features in a machine learning framework to predict the need of refactoring. We conduct a series of experiments on a publicly available annotated dataset of five software systems to investigate the performance of our proposed approach. In this proposed solution, ten different machine learning classifiers have been considered. In order to handle issues related to class imbalance, three different data sampling methods are also considered during implementation. *Conclusion :* Our analysis reveals that the mean accuracy for the SMOTE and RUSBoost data sampling technique is 98.47% respectively. The mean accuracy for the classifier AdaBoost is 98.16% and the mean accuracy for the classifier ANN+GD is 98.17% respectively. Hypothesis testing results reveals that the performance of different classifiers and data sampling techniques are statistically significant in nature.

## CCS CONCEPTS

• **Software and its engineering → Software post-development issues**; *Software creation and management*; • **Computing methodologies → Machine learning**.

## KEYWORDS

Empirical Software Engineering and Measurements (ESEM), Machine Learning, Software Refactoring, Software Maintenance, Source Code Analysis and Measurement

**ACM Reference Format:**
Lov Kumar, Shashank Mouli Satapathy, and Lalita Bhanu Murthy. 2019. Method Level Refactoring Prediction on Five Open Source Java Projects using Machine Learning Techniques. In *12th Innovations in Software Engineering Conference (formerly known as India Software Engineering Conference) (ISEC'19), February 14–16, 2019, Pune, India*, 10 pages. https://doi.org/10.1145/3299771.3299777


*ISEC'19, February 14–16, 2019, Pune, India*

ACM ISBN 978-1-4503-6215-3/19/02…$15.00
https://doi.org/10.1145/3299771.3299777

## 1 RESEARCH MOTIVATION AND AIM

Software Refactoring consists of changing the internal structure of the application or its design without changing its external behaviour or functionality [7, 16]. Refactoring is applied to improve the comprehensibility, complexity and maintainability of the software system [7, 14, 16]. The semantics of the code before and after refactoring remains the same. Martin Fowler has presented a list of refactoring on a website called as catalog[1] of refactoring. For example, Pull Up Method is a type of refactoring when the system has methods with identical results on subclasses. The process of refactoring consists of modifying classes, methods and variables in the application. It is technically challenging for software developers to detect elements or regions of a large complex system which requires refactoring.

*Which method(s) within a class to refactor?* is a problem encountered by software developers. Our research is motivated by the need to develop recommendation systems which can be integrated in the development environment and development processes of software engineers for suggesting methods in need of refactoring. There has been prior work on identifying classes and regions in a source code in need of refactoring (refer to the Section on Related Work). However, a large scale study involving several object-oriented Java based software systems on the usage of Machine Learning (ML) based techniques and using object-oriented source code metrics as features or predictors is relatively unexplored.

**Research Contributions:** In context to existing work, the work presented in this paper makes several novel and unique research contributions. The work presented in this paper is an extension to our previous work on class-level refactoring prediction [15]. In our previous work, we compute metrics at class-level and predict the need of refactoring at class-level [15]. In this work, we compute source code metrics at method-level and make predictions at method-level. To the best of our knowledge, ours is the *first study* on method level refactoring prediction on 5 open-source Java based projects i.e., antlr4, junit, mct, oryx, and titan using *10 different classifiers* i.e., Logistic Regression, NaiveBayes, BayesNet, RBFN, Multilayer Perceptron, Random Forest, AdaBoost, LogitBoost, ANN+GD, and ANN+LM, *25 source-code metrics* and *3 different data sampling techniques* i.e., SMOTE, over-sampling, RUSBoost.

[1] https://refactoring.com/catalog/

## 2 RELATED WORK

For identification of refactoring candidates, a class-based approach has been introduced by Zhao et al. [21] considering a chosen set of static source code metrics and furthermore, utilizing a weighted ranking method for predicting a list of classes, which require refactoring. Their study compared the performance of refactoring decision tool in order to reduce maintainability of class and proved that the use of tool helps in assisting the software team for the evaluation process significantly. Jehad Al Dallal [2] presented a measure and a prescient model to decisively identify, whether method(s) in a class needing move method refactoring and achieved prediction accuracy of more than 90%. The author considered the application of predictive model over seven object-oriented systems in order to empirically evaluate their performance. An in-depth analysis on the effects of refactoring over different internal quality attributes such as inheritance, complexity etc. have been presented by Alexander et al. [5]. They considered the history about different versions of twenty-three open source projects with more than 29,000 operations related to refactoring process. From the analysis of the results, it is observed that 94% of refactoring task applied to code having at least a single quality attribute. They also observed that 65% of refactoring task enhances related internal quality attributes and significant improvements are also observed to the quality attributes, when pure refactoring operation are applied. Kosker et al. [13] proposed an intelligent system by analyzing the code complexity in order to identify the class that require refactoring. Their approach is based on creating a machine learning model using Weighted NaÃŕve Bayes with InfoGain heuristic as the learner and conducted experiments on real world software system. The analysis of the predicted result proved that on an average, 13% of manual inspection effort is required to predict 82% of the classes in need of refactoring.

Jacek et al. [18]conducted an empirical study on open source project and observed that features of software evolution data can be considered to develop a model for predicting the refactoring during development phase. Their approach is based on development of the prediction models using logistic model, decision trees, nearest neighbor algorithms, and propositional rule learners by considering features of Version systems log data. They concluded the model developed for prediction refactoring of object-oriented systems using these features and classification techniques have high value of recall and precision. In this work they have also investigated different domains and derived some critical factors leading to refactoring in SDLC. Oscar et al. [4] proposed one technique called RIPE i.e., Refactoring Impact Prediction. This technique is used to predict the effect of refactoring operations on software quality. This technique supports 11 metrics and 12 refactoring operations and they concluded that this technique can be also used together with any tool that is used for refactoring recommendation. The analysis of the predicted result proved that RIPE on an average, 31% prediction for refactoring are correct.

Bavota et al. [3] conducted an empirical evaluation about the relationship between code smells and refactoring activities by mining more than 12000 refactoring operations. Results indicate that a total of 42% of refactoring task is implemented over codes influenced by code smells, where as only 7% of implemented tasks are capable of expeling code smells from influenced class. Kim et al. [11] investigated API-level refactoring process and procedures to improve number of bug fixes quantitatively considering three large open source projects. The implementation result suggested that the total number of bug fixes count have been improved as well as the the time required in order to fix bugs becomes less convincingly after API-level refactoring. OâĂŹKeeffe et al. [17] developed a tool to automatically refactor object-oriented software based on a predefined quality model in order improve its flexibility, reusability and understandability.

## 3 EXPERIMENTAL DATASET

The experimental dataset used in our study is freely and publicly available at tera-PROMISE Repository[2]. The tera-PROMISE online resource is a well-known software engineering research dataset repository consisting of experimental datasets on several engineering topics such as source code analysis, faults, effort estimation, refactoring, source code metrics and test generation [1]. We download the dataset from the tera-PROMISE repository for our experiments. This makes our work easily replicable and makes it easy for other researchers to compare or benchmark their approaches with our proposed method on the same dataset. The dataset used in our experiments is manually validated by the authors Kadar et al. [9, 10] who shared the dataset on tera-PROMISE. Kadar et al. create the source code metrics and the refactoring dataset for two subsequent releases of 7 well-known OSS (open source software) Java applications [9, 10]. The 7 Java-based OSS systems are available on GitHub[3] repository. Kadar et al. use the RefFinder tool [12] for identifying refactoring in the source code between two subsequent releases. They compute the source code metrics using the SourceMeter tool[4]. We use the method level metrics in the work presented in this paper and not the system level metrics. The metrics used in our previous work (refer to [15]) is at the class level and in this work we use the method level metrics of the same dataset. Table 1 shows the list of Java projects, Number of Methods (# NOM), Number of Refactored Methods (# NOMR), Number of Non-Refactored Methods (# NONMR),and Percentage of Refactored Methods (% RM). Table 1 reveals that the dataset is highly imbalanced as the % RM values are 0.13%, 0.19%, 0.52%, 0.75% and 1.21% respectively.

**Table 1: Experimental Data Set Description**

| Project | NOM | NORM | NONRM | %RM |
|---|---|---|---|---|
| antlr4 | 3298 | 40 | 3258 | 1.213 |
| junit | 2280 | 12 | 2268 | 0.526 |
| mct | 11683 | 16 | 11667 | 0.137 |
| oryx | 2507 | 19 | 2488 | 0.758 |
| titan | 8558 | 17 | 8541 | 0.199 |

[2] http://promise.site.uottawa.ca/SERepository/
[3] https://github.com/
[4] https://www.sourcemeter.com/

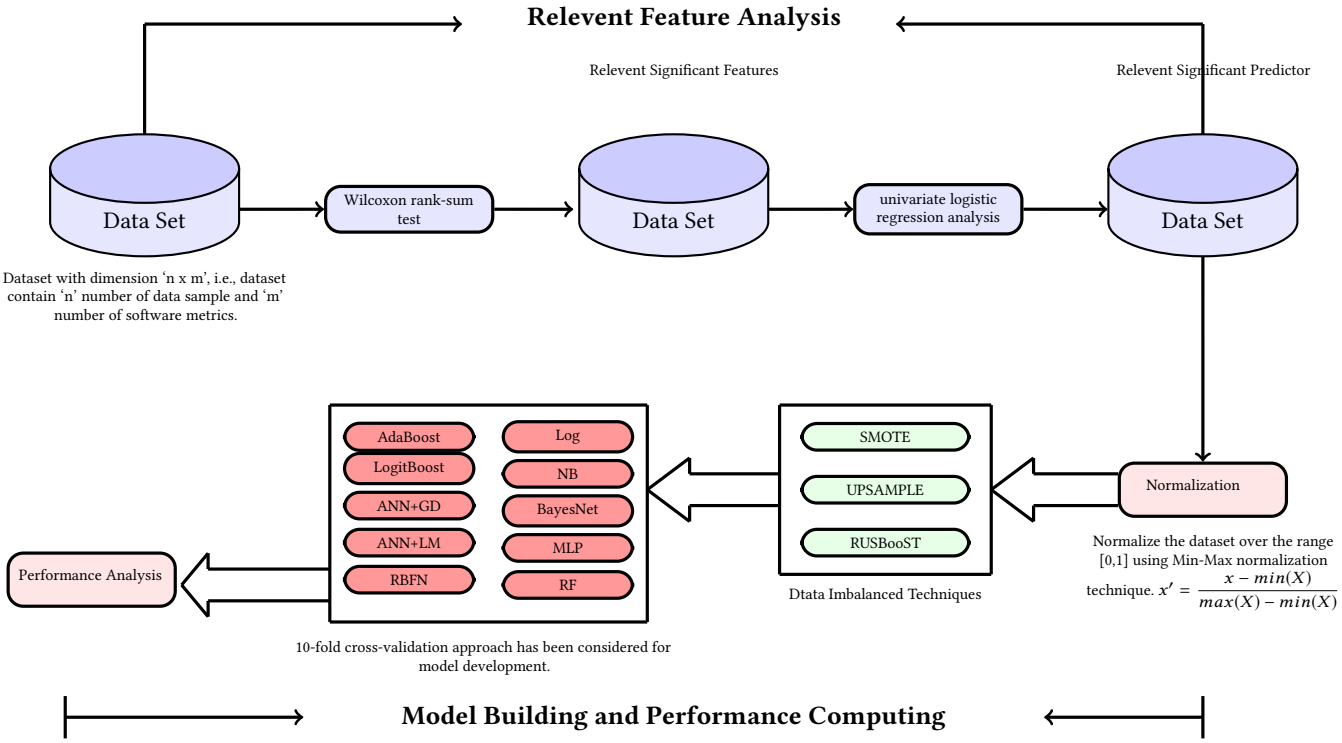



**Figure 1: Framework of proposed work**

## 4 SOLUTION APPROACH AND RESULTS

This section presents the solution approach followed in this experiment in order to predict code refactor at method level using various AI techniques. Figure 1 illustrates the proposed framework for development of method level refactoring prediction model for Java projects. From Figure 1, it can be clearly seen that the proposed approach is a multi-step process consisting of relevant features analysis using wilcoxon ran-sum test and univariate logistic regression analysis, feature scaling using min-max normalization technique, application of data sampling techniques to handle data imbalance problem, applying ten different classification techniques to train the models, and finally evaluation of proposed model using different performance parameters.

The first phase of the proposed solution deals with analyzing relevant features. During this phase, the considered dataset is pre-processed to extract relevant features. Initially, the Wilcoxon rank-sum test has been applied to handle uncertainty in the dataset and also for extracting relevant features. Subsequently, ULR analysis is applied to identify final set of source code metrics to be considered for further implementation. Second phase of proposed solution involves in model building process and assessment of performance of the proposed model. During the model building process, first of all the dataset is normalized through min-max normalization. As the considered dataset is imbalanced in nature, hence the dataset has been pre-processed to handle imbalanced learning result. The class imbalance issue of the dataset is addressed using Synthetic Minority Over-sampling Technique (SMOTE), UPSAMPLE and RUSBoost techniques. SMOTE technique is based on over-sampling approach in which "synthetic" examples are used for oversampling the minority class rather than over-sampling with replacement. UPSAMPLE is also used to improve the number of sample of minority class by inserting zeros between the samples. RUSBoost is hybrid approach of data sampling and boosting algorithm. The objective of RUSBoost is to improve the performance of models trained on skewed data. Then, 10-fold cross validation approach has been considered with ten different classification techniques to implement the proposed solution. Finally, the results obtained from each technique has been compared with the help of different performance measures in order to evaluate them.

### 4.1 Source Code Metrics

The source code metrics computed using the SourceMeter[5] for Java tool are considered in proposed solution. We make use of 25 different source code metrics such as McCabeâĂŹs Cyclomatic Complexity (McCC), Clone Classes (CCL), Clone Complexity (CCO), Total Lines of Code (TLOC), Total Number of Statements (TNOS), Number of Incoming Invocations (NII) and Number of Outgoing Invocations (NOI). The list of all the metrics shown in Figure 2 is provided on the SourceMeter tool website. The source code metrics are used as features or independent variables for the machine learning algorithms. The input to the machine learning algorithms are the source code metrics and the output is a binary class (whether a

[5]https://www.sourcemeter.com/resources/java/

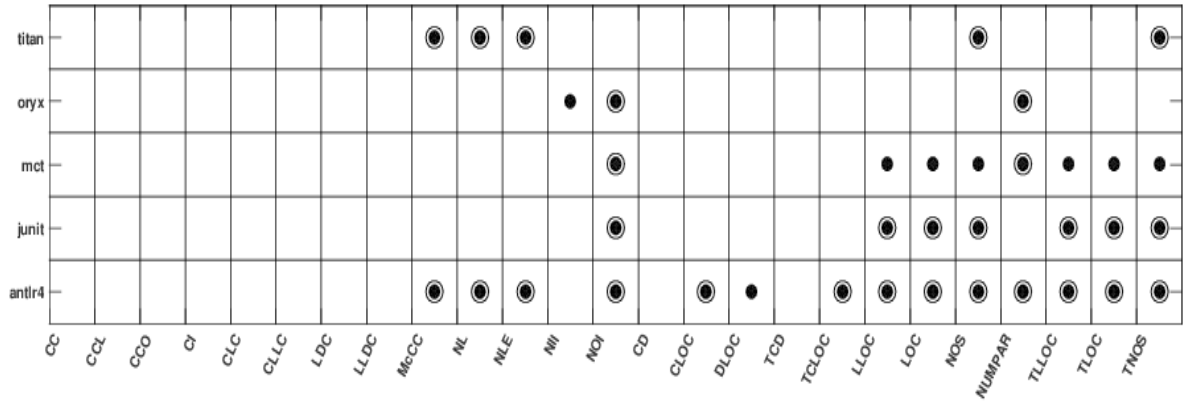



**Figure 2: Selected set of metrics using Wilcoxon rank-sum test and ULR analysis**

method is need of refactoring or now). We conduct a metrics selection using Wilcoxon rank-sum test and ULR (Univariate Logistic Regression). The graphs in Figure 2 are represented using different symbols: filled circle: source code metrics selected using Wilcoxon rank-sum test, black circle with bold circle: source code metrics selected using Wilcoxon rank-sum test and ULR analysis. We perform metrics selection to remove metrics which are not relevant. Through metrics selection, we identify relevant metrics for the task of refactoring prediction. From Figure 2, it can be seen that NOI, NOS, NUMPAR, TNOS are commonly found significant and relevant metrics related for refactoring prediction of methods in most of the software. From Figure 2, it can be also seen that CC, CCL, CCO, CI, CLC, CLLC, LDC, LLDC, TCD metrics are not significant and relevant metrics for refactoring prediction of methods.

**Table 2: Performance Results (SMOTE)**

| | **Accuracy** | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | **Log** | **NB** | **BayesNet** | **RBFN** | **MLP** | **RF** | **AdaBoost** | **LogitBoost** | **ANN+GD** | **ANN+LM** |
| antlr4 | 98 | 92.8 | 83.8 | 99 | 97.8 | 98.4 | 98 | 98.2 | 98.8 | 98.2 |
| junit | 99.4 | 92.8 | 94.8 | 99.6 | 98.8 | 99.2 | 99.6 | 99.4 | 99.4 | 99 |
| mct | 100 | 98.8 | 100 | 100 | 100 | 100 | 100 | 100 | 99.8 | 100 |
| oryx | 98.8 | 96.2 | 98.8 | 98.8 | 98.8 | 98.8 | 98.8 | 98.6 | 99 | 98.6 |
| titan | 100 | 95.8 | 99.2 | 99.8 | 100 | 100 | 100 | 100 | 99.8 | 100 |
| | **F-Measure** | | | | | | | | | |
| antlr4 | 0.99 | 0.96 | 0.91 | 0.99 | 0.99 | 0.99 | 0.99 | 0.99 | 0.99 | 0.99 |
| junit | 1 | 0.96 | 0.97 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| mct | 1 | 0.99 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| oryx | 0.99 | 0.98 | 1 | 0.99 | 0.99 | 0.99 | 0.99 | 0.99 | 1 | 0.99 |
| titan | 1 | 0.98 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| | **AUC** | | | | | | | | | |
| antlr4 | 0.64 | 0.67 | 0.7 | 0.54 | 0.68 | 0.68 | 0.7 | 0.69 | 0.51 | 0.71 |
| junit | 0.5 | 0.6 | 0.48 | 0.5 | 0.5 | 0.5 | 0.5 | 0.5 | 0.53 | 0.5 |
| mct | 0.5 | 0.53 | 0.5 | 0.5 | 0.5 | 0.5 | 0.5 | 0.5 | 0.5 | 0.5 |
| oryx | 0.53 | 0.65 | 0.52 | 0.5 | 0.5 | 0.52 | 0.5 | 0.55 | 0.53 | 0.55 |
| titan | 0.53 | 0.59 | 0.53 | 0.52 | 0.53 | 0.5 | 0.5 | 0.5 | 0.53 | 0.53 |

**Table 3: Performance Results (UPSAMPLE)**

| | Accuracy | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | **Log** | **NB** | **BayesNet** | **RBFN** | **MLP** | **RF** | **AdaBoost** | **LogitBoost** | **ANN+GD** | **ANN+LM** |
| antlr4 | 86.6 | 87.4 | 61 | 81.2 | 87.4 | 87.2 | 91.8 | 85.4 | 91.8 | 89.2 |
| junit | 95 | 88.2 | 99.6 | 94.4 | 94.6 | 86.2 | 95.4 | 90.8 | 96 | 93.6 |
| mct | 98.8 | 96 | 100 | 97.2 | 98 | 97.2 | 99.4 | 99 | 98.8 | 98.4 |
| oryx | 93.2 | 94 | 98.6 | 93 | 91.6 | 89.2 | 94 | 93.4 | 91.8 | 94 |
| titan | 98.8 | 94.8 | 99.6 | 99.4 | 99 | 99 | 99.2 | 99.2 | 100 | 98.4 |
| | **F-Measure** | | | | | | | | | |
| antlr4 | 0.93 | 0.93 | 0.75 | 0.89 | 0.93 | 0.93 | 0.96 | 0.92 | 0.96 | 0.94 |
| junit | 0.97 | 0.94 | 1 | 0.97 | 0.97 | 0.92 | 0.98 | 0.95 | 0.98 | 0.97 |
| mct | 0.99 | 0.98 | 1 | 0.99 | 0.99 | 0.99 | 1 | 1 | 0.99 | 0.99 |
| oryx | 0.96 | 0.97 | 1 | 0.96 | 0.96 | 0.94 | 0.97 | 0.97 | 0.95 | 0.97 |
| titan | 1 | 0.97 | 1 | 1 | 1 | 0.99 | 1 | 1 | 1 | 0.99 |
| | **AUC** | | | | | | | | | |
| antlr4 | 0.75 | 0.68 | 0.74 | 0.75 | 0.75 | 0.79 | 0.75 | 0.79 | 0.67 | 0.74 |
| junit | 0.69 | 0.64 | 0.5 | 0.51 | 0.66 | 0.63 | 0.56 | 0.59 | 0.48 | 0.57 |
| mct | 0.55 | 0.53 | 0.5 | 0.57 | 0.61 | 0.54 | 0.5 | 0.5 | 0.55 | 0.56 |
| oryx | 0.69 | 0.65 | 0.5 | 0.64 | 0.68 | 0.74 | 0.65 | 0.67 | 0.49 | 0.62 |
| titan | 0.55 | 0.58 | 0.5 | 0.57 | 0.55 | 0.55 | 0.57 | 0.57 | 0.5 | 0.54 |

**Table 4: Performance Results (RUSBOOST)**

| | Accuracy | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | **Log** | **NB** | **BayesNet** | **RBFN** | **MLP** | **RF** | **AdaBoost** | **LogitBoost** | **ANN+GD** | **ANN+LM** |
| antlr4 | 98 | 92.4 | 84.6 | 99 | 98 | 98.8 | 97.8 | 98.2 | 99 | 98.4 |
| junit | 99.6 | 92.8 | 91.8 | 99.6 | 99.2 | 99 | 99.6 | 99.4 | 99.4 | 99.2 |
| mct | 100 | 99 | 100 | 100 | 100 | 100 | 100 | 100 | 100 | 100 |
| oryx | 99 | 96 | 99 | 98.6 | 98.8 | 98.8 | 98.8 | 99 | 99 | 99 |
| titan | 100 | 95.6 | 99 | 100 | 100 | 100 | 100 | 100 | 100 | 100 |
| | **F-Measure** | | | | | | | | | |
| antlr4 | 0.99 | 0.96 | 0.92 | 0.99 | 0.99 | 0.99 | 0.99 | 0.99 | 0.99 | 0.99 |
| junit | 1 | 0.96 | 0.96 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| mct | 1 | 0.99 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| oryx | 1 | 0.98 | 1 | 0.99 | 0.99 | 0.99 | 1 | 0.99 | 1 | 0.99 |
| titan | 1 | 0.98 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| | **AUC** | | | | | | | | | |
| antlr4 | 0.65 | 0.69 | 0.69 | 0.52 | 0.68 | 0.72 | 0.68 | 0.69 | 0.52 | 0.69 |
| junit | 0.5 | 0.6 | 0.56 | 0.5 | 0.5 | 0.5 | 0.5 | 0.5 | 0.5 | 0.5 |
| mct | 0.5 | 0.53 | 0.5 | 0.5 | 0.5 | 0.5 | 0.5 | 0.5 | 0.5 | 0.5 |
| oryx | 0.52 | 0.64 | 0.56 | 0.55 | 0.52 | 0.55 | 0.5 | 0.56 | 0.5 | 0.53 |
| titan | 0.5 | 0.59 | 0.57 | 0.52 | 0.5 | 0.5 | 0.5 | 0.5 | 0.55 | 0.5 |

## 4.2 Imbalance Learning Results

One of the technical challenges in building a classifier for refactoring candidate prediction is that the dataset is imbalanced. We address the class imbalance issue using three techniques: SMOTE, RUSBoost and UPSAMPLE. UPSAMPLE is an oversampling technique in which the minority class is oversampled. SMOTE (Synthetic Minority Over-sampling Technique) algorithm combines

under-sampling of the majority class in conjunction with a specialized type of minority class over-sampling [6]. We also use a method called as RUSBoost proposed by Seiffert et al. also used for addressing the class-imbalance problem similar to encountered in our dataset [19, 20]. RUSBoost is a well-known technique which integrates both data sampling and boosting [19, 20]. RUSBoost makes use of a technique called as boosting which is also used by widely used algorithms like AdaBoost. It consists of creating an ensemble of predictive models for building a collection of weighted classifiers for classifying the test instances [19, 20]. In particular, the RUSBoost technique has been shown to perform positve on skewed training data [19, 20].

Table 2, 3 and 4 displays the detailed performance results obtained after applying all the three imbalanced learning techniques (UPSAMPLE, SMOTE and RUSBoost). Table 2, 3 and 4 shows the results for all the classification algorithms and dataset combinations. The performance results are shown in-terms of accuracy, f-measure and AUC. The experimental results shows that techniques like SMOTE and RUSBoost gives encouraging results. The results in Tables 2, 3 and 4 shows variations in the performance depending on the classifier. For example, SMOTE results in better performance in combination with RBFN, AdaBoost and LogitBoost in comparison to NB and ANN+GD. The performance of RUSBoost in combination with ANN+GD is much higher than the performance of SMOTE in combination with ANN+GD. Overall RUSBoost performs better than UPSAMPLE and SMOTE.

## 4.3 Performance Visualization using Boxplots

Figure 3 shows multiple boxplots for analyzing the degree of spread or dispersion, outliers, skewness, interquartile range in the accuracy, f-measure and AUC performance metrics for the classifiers and data sampling techniques. The red line present in boxplot as shown in Figure 3 signifies the median value in order to divide the box into two segments. Figure 3 also presents the variations observed in the median values for the classifiers and data sampling techniques. The median value for the accuracy and f-measure of the NB classifier is lower than all other classifiers and similarly the median values for the UPSAMPLE is lower than SMOTE and RUSBoost. By analyzing the results shown in Figure 3, it can be inferred that the interquartile range representing the middle box of the accuracy boxplot (middle 50% of the values of the given variable) for BayesNet and UPSAMPLE is highest with respect to the corresponding values of other techniques. The length of various boxplots in Figure 3 from minimum to maximum varies significantly. We observe that for the AUC boxplots, RF and LogitBoost boxplots are taller in comparison to the boxplots of other classifiers. The whiskers in the boxplot of Figure 3 shows minimum and maximum values and as shown in Figure 3, the maximum AUC value for UPSAMPLE is much higher than the maximum AUC value for SMOTE and RUSBoost.

## 4.4 Descriptive Statistics in-terms of Accuracy, F-Measure and AUC

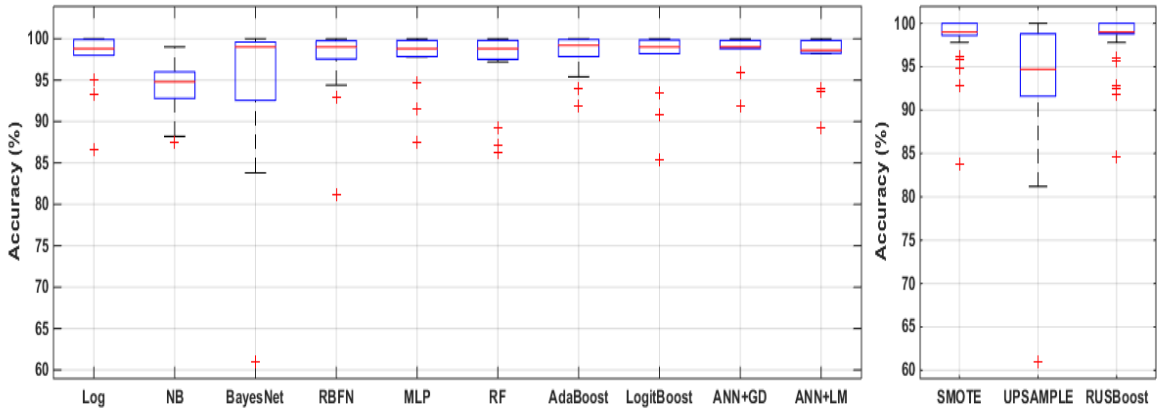
Statistical description about the overall performance of the 13 different techniques in-terms of accuracy has been presented in Table 5. While computing the overall performance of a particular technique, we take the average of 13 values consisting of the remaining techniques. Every value consist of the average performance for the 5 dataset. Hence every value in the Tables 5, 6 and 8 is computed by taking the average of $12 \times 5 = 60$ values. Table 5 reveals that the mean accuracy for the SMOTE and RUSBoost data sampling technique is 98.47% respectively. The mean accuracy for AdaBoost is 98.16% and the mean accuracy for ANN+GD is 98.17% respectively. From Table 5, we infer that AdaBoost and ANN+GD gives the best performance amongst the 10 classifiers. Amongst the 3 data sampling technique, SMOTE and RUSBoost outperforms UPSAMPLE by 5%. Similarly, we observe that techniques like AdaBoost and ANN+GD outperforms BayesNet by 5%. Table 6 reveals that the mean value of the F-Measure for SMOTE and RUSBoost is 0.99 and the mean value of the F-measure for UPSAMPLE is 0.97. From Table 6, we infer that the standard deviation for the 13 different technique varies from 0.01 to 0.07. Table 8 displays the AUC values for all the 13 different techniques. Each AUC value is computed by taking an average of 60 executions of the predictive model.

**Table 5: Descriptive Statistics of the Overall Performance of a Technique across All Datasets in term of Accuracy**

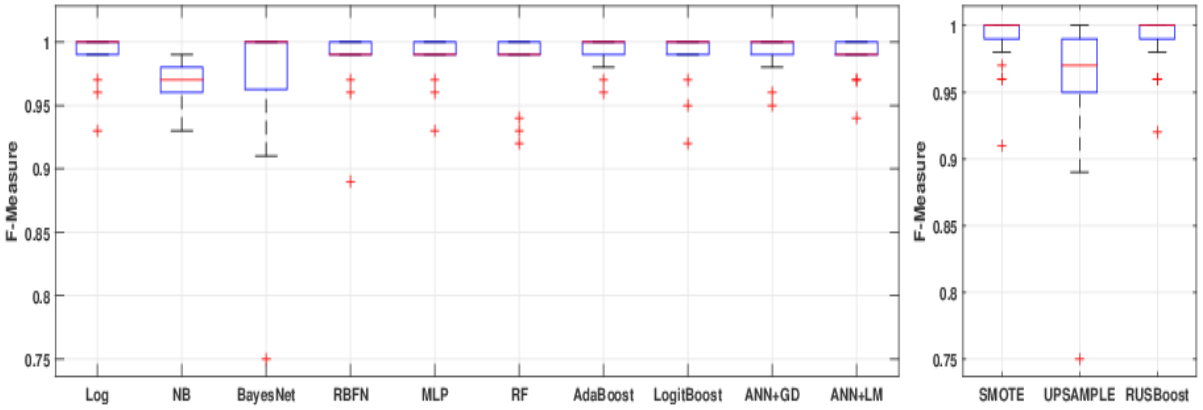
| | Accuracy | | | | | | |
|---|---|---|---|---|---|---|---|
| | **Min** | **Max** | **Mean** | **Median** | **Std Dev** | **Q1** | **Q3** |
| Log | 86.60 | 100.00 | 97.68 | 98.80 | 3.62 | 98.00 | 99.90 |
| NB | 87.40 | 99.00 | 94.17 | 94.80 | 3.28 | 92.80 | 96.00 |
| BayesNet | 61.00 | 100.00 | 93.99 | 99.00 | 10.62 | 92.55 | 99.60 |
| RBFN | 81.20 | 100.00 | 97.31 | 99.00 | 4.92 | 97.55 | 99.75 |
| MLP | 87.40 | 100.00 | 97.47 | 98.80 | 3.60 | 97.85 | 99.80 |
| RF | 86.20 | 100.00 | 96.79 | 98.80 | 4.88 | 97.50 | 99.80 |
| AdaBoost | 91.80 | 100.00 | 98.16 | 99.20 | 2.49 | 97.85 | 99.90 |
| LogitBoost | 85.40 | 100.00 | 97.37 | 99.00 | 4.22 | 98.20 | 99.85 |
| ANN+GD | 91.80 | 100.00 | 98.17 | 99.00 | 2.77 | 98.80 | 99.80 |
| ANN+LM | 89.20 | 100.00 | 97.73 | 98.60 | 3.07 | 98.25 | 99.80 |
| SMOTE | 83.80 | 100.00 | 98.47 | 99.00 | 2.68 | 98.60 | 100.00 |
| UPSAMPLE | 61.00 | 100.00 | 93.72 | 94.70 | 6.68 | 91.60 | 98.80 |
| RUSBoost | 84.60 | 100.00 | 98.47 | 99.00 | 2.75 | 98.80 | 100.00 |

**Table 6: Descriptive Statistics of the Overall Performance of a Technique across All Datasets in-terms of F-Measure**

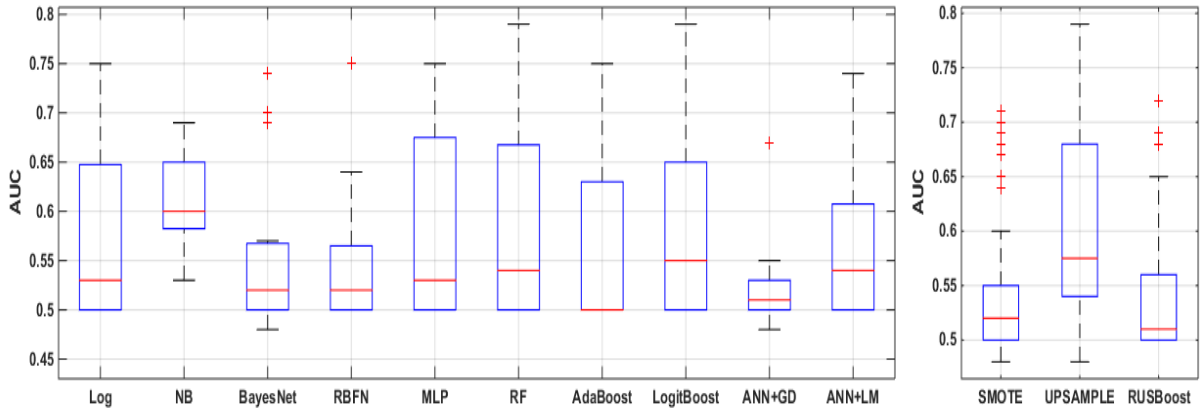
| | F-Measure | | | | | | |
|---|---|---|---|---|---|---|---|
| | **Min** | **Max** | **Mean** | **Median** | **Std Dev** | **Q1** | **Q3** |
| Log | 0.93 | 1.00 | 0.99 | 1.00 | 0.02 | 0.99 | 1.00 |
| NB | 0.93 | 0.99 | 0.97 | 0.97 | 0.02 | 0.96 | 0.98 |
| BayesNet | 0.75 | 1.00 | 0.97 | 1.00 | 0.07 | 0.96 | 1.00 |
| RBFN | 0.89 | 1.00 | 0.98 | 0.99 | 0.03 | 0.99 | 1.00 |
| MLP | 0.93 | 1.00 | 0.99 | 0.99 | 0.02 | 0.99 | 1.00 |
| RF | 0.92 | 1.00 | 0.98 | 0.99 | 0.03 | 0.99 | 1.00 |
| AdaBoost | 0.96 | 1.00 | 0.99 | 1.00 | 0.01 | 0.99 | 1.00 |
| LogitBoost | 0.92 | 1.00 | 0.99 | 1.00 | 0.02 | 0.99 | 1.00 |
| ANN+GD | 0.95 | 1.00 | 0.99 | 1.00 | 0.02 | 0.99 | 1.00 |
| ANN+LM | 0.94 | 1.00 | 0.99 | 0.99 | 0.02 | 0.99 | 1.00 |
| SMOTE | 0.91 | 1.00 | 0.99 | 1.00 | 0.02 | 0.99 | 1.00 |
| UPSAMPLE | 0.75 | 1.00 | 0.97 | 0.97 | 0.04 | 0.95 | 0.99 |
| RUSBoost | 0.92 | 1.00 | 0.99 | 1.00 | 0.01 | 0.99 | 1.00 |




**(3.1) Accuracy (%)**




**(3.2) F-Measure**




**(3.3) AUC**

**Figure 3: Performance Visualization using Boxplots**

**Table 8: Descriptive Statistics of the Overall Performance of a Technique across All Datasets in-term of AUC**

| | AUC | | | | | | |
|---|---|---|---|---|---|---|---|
| | **Min** | **Max** | **Mean** | **Median** | **Std Dev** | **Q1** | **Q3** |
| Log | 0.50 | 0.75 | 0.57 | 0.53 | 0.09 | 0.50 | 0.65 |
| NB | 0.53 | 0.69 | 0.61 | 0.60 | 0.05 | 0.58 | 0.65 |
| BayesNet | 0.48 | 0.74 | 0.56 | 0.52 | 0.08 | 0.50 | 0.57 |
| RBFN | 0.50 | 0.75 | 0.55 | 0.52 | 0.07 | 0.50 | 0.57 |
| MLP | 0.50 | 0.75 | 0.58 | 0.53 | 0.09 | 0.50 | 0.68 |
| RF | 0.50 | 0.79 | 0.58 | 0.54 | 0.10 | 0.50 | 0.67 |
| AdaBoost | 0.50 | 0.75 | 0.56 | 0.50 | 0.09 | 0.50 | 0.63 |
| LogitBoost | 0.50 | 0.79 | 0.57 | 0.55 | 0.09 | 0.50 | 0.65 |
| ANN+GD | 0.48 | 0.67 | 0.52 | 0.51 | 0.05 | 0.50 | 0.53 |
| ANN+LM | 0.50 | 0.74 | 0.57 | 0.54 | 0.08 | 0.50 | 0.61 |
| SMOTE | 0.48 | 0.71 | 0.55 | 0.52 | 0.07 | 0.50 | 0.55 |
| UPSAMPLE | 0.48 | 0.79 | 0.61 | 0.58 | 0.09 | 0.54 | 0.68 |
| RUSBoost | 0.50 | 0.72 | 0.55 | 0.51 | 0.07 | 0.50 | 0.56 |

## 4.5 Bonferroni Correction and Holm Method

Table 7 shows the results of the Holm-Bonferroni method. We compute the p-value and the rank order of the p-value in-order to reduce the Type II errors (false negative). We do not assign a raw p-value, which is stringent and strict for all the comparisons. Different pairs are shown in Table 7 based on which the hypothesis testing results are presented in Figure 4. We use the Wilcoxon test with Holm-BonferroniMethod for comparative analysis between different approaches because the Wilcoxon test without Bonferroni correction does not take into account the family-wise errors [8]. In Table 7, in the Holm-Bonferroni Method, we adjust the significance cutoff at $\frac{0.05}{n-\text{rank of pair}+1}$, where $n$ is number of different pairs (in our case it is 10 different techniques : n=$^{10 technique}C_2 = 10 * 9/2 = 45$. Table 7 reveals that the rank order for the NB and BayesNet comparison is 24 and the p-value is 0.046. The adjusted significant cut-off is 0.0023. Similarly, the rank order for LogitBoost and ANN+GD comparison is 20 and the p-value is 0.022.

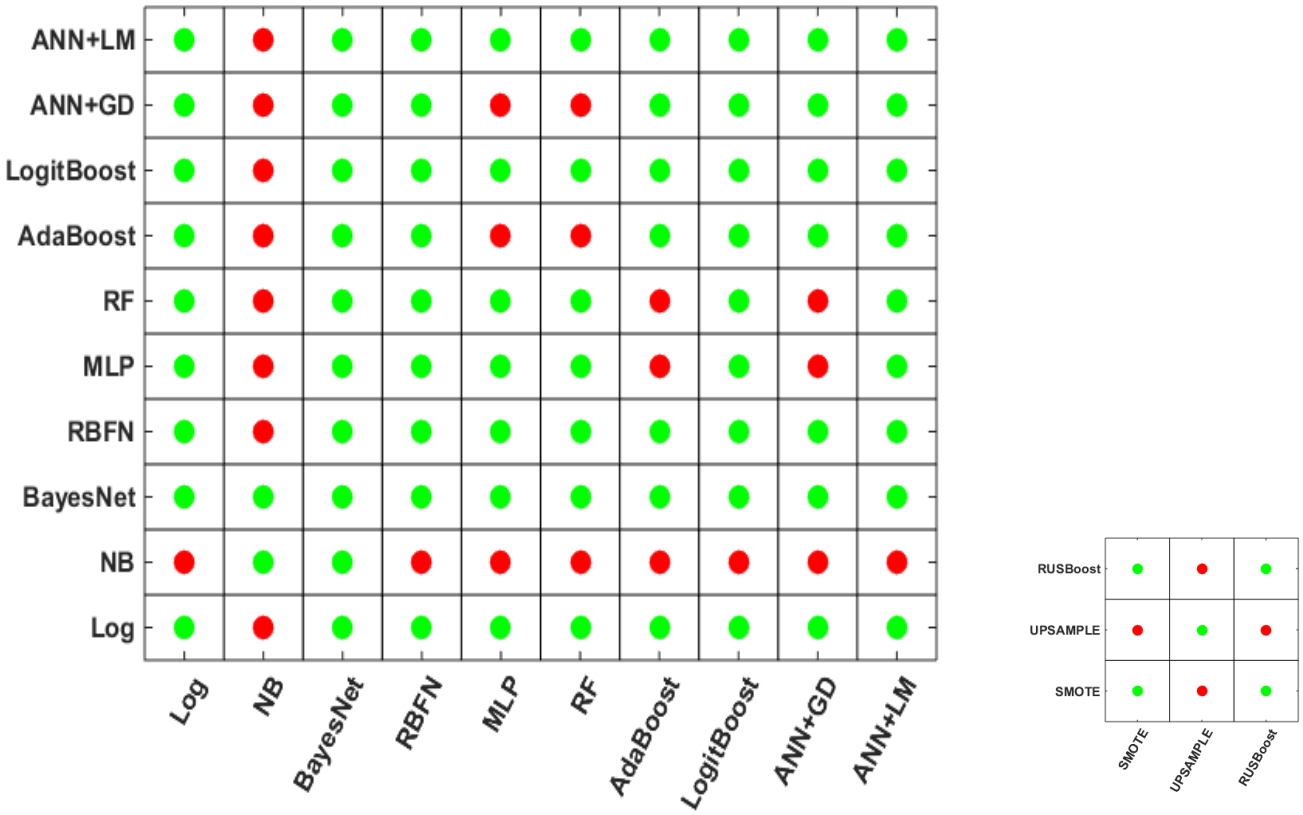
**Table 7: Bonferroni Correction and Holm Method**

| Unique Pair | P-value | Rank | Adjusted significance cutoff | Unique Pair | P-value | Rank | Adjusted significance cutoff |
|---|---|---|---|---|---|---|---|
| (Log,NB) | 0.000 | 4 | 0.0012 | (RBFN,MLP) | 0.328 | 37 | 0.0056 |
| (Log,BayesNet) | 0.045 | 22 | 0.0021 | (RBFN,RF) | 0.147 | 32 | 0.0036 |
| (Log,RBFN) | 0.975 | 45 | 0.0500 | (RBFN,AdaBoost) | 0.148 | 33 | 0.0038 |
| (Log,MLP) | 0.479 | 39 | 0.0071 | (RBFN,LogitBoost) | 0.434 | 38 | 0.0063 |
| (Log,RF) | 0.003 | 14 | 0.0016 | (RBFN,ANN+GD) | 0.116 | 30 | 0.0031 |
| (Log,AdaBoost) | 0.003 | 16 | 0.0017 | (RBFN,ANN+LM) | 0.704 | 41 | 0.0100 |
| (Log,LogitBoost) | 0.510 | 40 | 0.0083 | (MLP,RF) | 0.294 | 36 | 0.0050 |
| (Log,ANN+GD) | 0.002 | 13 | 0.0015 | (MLP,AdaBoost) | 0.000 | 12 | 0.0015 |
| (Log,ANN+LM) | 0.782 | 42 | 0.0125 | (MLP,LogitBoost) | 0.128 | 31 | 0.0033 |
| (NB,BayesNet) | 0.046 | 24 | 0.0023 | (MLP,ANN+GD) | 0.000 | 11 | 0.0014 |
| (NB,RBFN) | 0.000 | 7 | 0.0013 | (MLP,ANN+LM) | 0.186 | 34 | 0.0042 |
| (NB,MLP) | 0.000 | 3 | 0.0012 | (RF,AdaBoost) | 0.000 | 10 | 0.0014 |
| (NB,RF) | 0.000 | 8 | 0.0013 | (RF,LogitBoost) | 0.021 | 19 | 0.0019 |
| (NB,AdaBoost) | 0.000 | 1 | 0.0011 | (RF,ANN+GD) | 0.000 | 9 | 0.0014 |
| (NB,LogitBoost) | 0.000 | 6 | 0.0013 | (RF,ANN+LM) | 0.110 | 29 | 0.0029 |
| (NB,ANN+GD) | 0.000 | 2 | 0.0011 | (AdaBoost,LogitBoost) | 0.007 | 18 | 0.0018 |
| (NB,ANN+LM) | 0.000 | 5 | 0.0012 | (AdaBoost,ANN+GD) | 0.880 | 43 | 0.0167 |
| (BayesNet,RBFN) | 0.053 | 26 | 0.0025 | (AdaBoost,ANN+LM) | 0.048 | 25 | 0.0024 |
| (BayesNet,MLP) | 0.073 | 28 | 0.0028 | (LogitBoost,ANN+GD) | 0.022 | 20 | 0.0019 |
| (BayesNet,RF) | 0.250 | 35 | 0.0045 | (LogitBoost,ANN+LM) | 0.951 | 44 | 0.0250 |
| (BayesNet,AdaBoost) | 0.003 | 15 | 0.0016 | (ANN+GD,ANN+LM) | 0.045 | 23 | 0.0022 |
| (BayesNet,LogitBoost) | 0.054 | 27 | 0.0026 | (SMOTE,UPSAMPLE) | 0.000 | 2 | 0.0250 |
| (BayesNet,ANN+GD) | 0.006 | 17 | 0.0017 | (SMOTE,RUSBoost) | 0.116 | 3 | 0.0500 |
| (BayesNet,ANN+LM) | 0.024 | 21 | 0.0020 | (UPSAMPLE,RUSBoost) | 0.000 | 1 | 0.0167 |

### 4.6 Hypothesis Testing Results

Figure 4 consists of two sub-figures. The first sub-figure consists of 100 cells as there are 10 classifiers. The second sub-figure consists of 9 cells as there are 3 data sampling techniques. We make pairwise comparisons between the classifiers and data sampling techniques. For example, we compare the performance of NB classifier with the performance of 9 other classifiers. Similarly, we compare the performance of SMOTE with two other data sampling technique UPSAMPLE and RUSBoost. Our null hypothesis $H_0$ is that there is no different between the given two techniques and our alternate hypothesis $H_A$ is that there is a different between the performance of the given two techniques. A red dot signifies the rejection of null hypothesis, where as a green dot signifies acceptance of null hypothesis. A red dot indicates that there is a statistically significant difference between the performances of given two techniques and the difference is not by chance. Figure 4 reveals that the null hypothesis is rejected for the following 12 cases. For all the remaining combinations, the null hypothesis is accepted. We list only the cases for which the null hypothesis is rejected as the number of cases for which the null hypothesis is accepted is large as there are 10 classification techniques.

(1) $H_0$ There is no statistically significant different between NB and Log.
(2) $H_0$ There is no statistically significant different between NB and RBFN.
(3) $H_0$ There is no statistically significant different between NB and MLP.
(4) $H_0$ There is no statistically significant different between NB and RF.
(5) $H_0$ There is no statistically significant different between NB and AdaBoost.
(6) $H_0$ There is no statistically significant different between NB and LogitBoost.
(7) $H_0$ There is no statistically significant different between NB and ANN+GD.
(8) $H_0$ There is no statistically significant different between NB and ANN+LM.
(9) $H_0$ There is no statistically significant different between MLP and AdaBoost.
(10) $H_0$ There is no statistically significant different between NB and ANN+GD.
(11) $H_0$ There is no statistically significant different between RF and AdaBoost.




(4.1) Classification Techniques

(4.2) Imbalance Techniques

**Figure 4: Hypothesis Testing Results**

(12) $H_0$ There is no statistically significant different between RF and ANN+GD.

Figure 4 reveals that the null hypothesis is rejected for the following two cases:

(1) $H_0$ There is no statistically significant different between UPSAMPLE and SMOTE.
(2) $H_0$ There is no statistically significant different between UPSAMPLE and RUSBoost.

Figure 4 reveals that the null hypothesis is accepted for the following case:

(1) $H_0$ There is no statistically significant different between RUSBoost and SMOTE.

## 5 CONCLUSION

We use the source code metrics computed using the SourceMeter[6] for Java tool. We make use of 25 different source code metrics such as McCabeâĂŹs Cyclomatic Complexity (McCC), Clone Classes (CCL), Clone Complexity (CCO), Total Lines of Code (TLOC), Total Number of Statements (TNOS), Number of Incoming Invocations (NII) and Number of Outgoing Invocations (NOI). The list of all the metrics shown in Figure 2 is provided on the SourceMeter tool website. The source code metrics are used as features or independent variables for the machine learning algorithms. The input to the machine learning algorithms are the source code metrics and the output is a binary class (whether a method is need of refactoring or now). We conduct a metrics selection using Wilcoxon rank-sum test and ULR (Univariate Logistic Regression). The graphs in Figure 2 are represented using different symbols: filled circle: source code metrics selected using Wilcoxon rank-sum test, black circle with bold circle: source code metrics selected using Wilcoxon rank-sum test and ULR analysis. We perform metrics selection to remove metrics which are not relevant. Through metrics selection, we identify relevant metrics for the task of refactoring prediction. We observe that there is a statistically significant difference between the performance of some of the classifiers and imbalance learning techniques. Overall, from our experiments we can conclude that method level refactoring prediction using source code metrics and machine learning classifier is possible. The analysis of the obained results revealed that for data sampling process SMOTE and RUSBoost outperforms other techniques, whereas the performance of AdaBoost and ANN+GD classifiers outperform others for the considered dataset.

## REFERENCES

[1] 2015. The Promise Repository of Empirical Software Engineering Data.

[6]https://www.sourcemeter.com/resources/java/

[2] Jehad Al Dallal. 2017. Predicting move method refactoring opportunities in object-oriented code. *Information and Software Technology* 92 (2017), 105–120.
[3] Gabriele Bavota, Andrea De Lucia, Massimiliano Di Penta, Rocco Oliveto, and Fabio Palomba. 2015. An experimental investigation on the innate relationship between quality and refactoring. *Journal of Systems and Software* 107 (2015), 1–14.
[4] Oscar Chaparro, Gabriele Bavota, Andrian Marcus, and Massimiliano Di Penta. 2014. On the impact of refactoring operations on code quality metrics. In *Software Maintenance and Evolution (ICSME), 2014 IEEE International Conference on*. IEEE, 456–460.
[5] Alexander Chávez, Isabella Ferreira, Eduardo Fernandes, Diego Cedrim, and Alessandro Garcia. 2017. How does refactoring affect internal quality attributes?: A multi-project study. In *Proceedings of the 31st Brazilian Symposium on Software Engineering*. ACM, 74–83.
[6] Nitesh V Chawla, Kevin W Bowyer, Lawrence O Hall, and W Philip Kegelmeyer. 2002. SMOTE: synthetic minority over-sampling technique. *Journal of artificial intelligence research* 16 (2002), 321–357.
[7] Martin Fowler and Kent Beck. 1999. *Refactoring: improving the design of existing code*. Addison-Wesley Professional.
[8] Yosef Hochberg and Yoav Benjamini. 1990. More powerful procedures for multiple significance testing. *Statistics in medicine* 9, 7 (1990), 811–818.
[9] István Kádár, Péter Hegedus, Rudolf Ferenc, and Tibor Gyimóthy. 2016. A code refactoring dataset and its assessment regarding software maintainability. In *Software Analysis, Evolution, and Reengineering (SANER), 2016 IEEE 23rd International Conference on*, Vol. 1. IEEE, 599–603.
[10] István Kádár, Péter Hegedűs, Rudolf Ferenc, and Tibor Gyimóthy. 2016. A Manually Validated Code Refactoring Dataset and Its Assessment Regarding Software Maintainability. In *Proceedings of the The 12th International Conference on Predictive Models and Data Analytics in Software Engineering*. ACM, 10.
[11] Miryung Kim, Dongxiang Cai, and Sunghun Kim. 2011. An empirical investigation into the role of API-level refactorings during software evolution. In *Proceedings of the 33rd International Conference on Software Engineering*. ACM, 151–160.
[12] Miryung Kim, Matthew Gee, Alex Loh, and Napol Rachatasumrit. 2010. RefFinder: a refactoring reconstruction tool based on logic query templates. In *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*. ACM, 371–372.
[13] Yasemin Kosker, Burak Turhan, and Ayse Bener. 2009. An expert system for determining candidate software classes for refactoring. *Expert Systems with Applications* 36, 6 (2009), 10000–10003.
[14] Lov Kumar, Shashank Mouli Satapathy, and Aneesh Krishna. 2018. Application of SMOTE and LSSVM with Various Kernels for Predicting Refactoring at Method Level. In *Neural Information Processing*, Long Cheng, Andrew Chi Sing Leung, and Seiichi Ozawa (Eds.). Springer International Publishing, Cham, 150–161.
[15] L. Kumar and A. Sureka. 2017. Application of LSSVM and SMOTE on Seven Open Source Projects for Predicting Refactoring at Class Level. In *2017 24th Asia-Pacific Software Engineering Conference (APSEC)*. 90–99.
[16] Tom Mens and Tom Tourwé. 2004. A survey of software refactoring. *IEEE Transactions on software engineering* 30, 2 (2004), 126–139.
[17] Mark OâĂŹKeeffe and Mel O Cinnéide. 2008. Search-based refactoring for software maintenance. *Journal of Systems and Software* 81, 4 (2008), 502–516.
[18] Jacek Ratzinger, Thomas Sigmund, Peter Vorburger, and Harald Gall. 2007. Mining software evolution to predict refactoring. In *Empirical Software Engineering and Measurement, 2007. ESEM 2007. First International Symposium on*. IEEE, 354–363.
[19] Chris Seiffert, Taghi M Khoshgoftaar, Jason Van Hulse, and Amri Napolitano. 2008. RUSBoost: Improving classification performance when training data is skewed. In *Pattern Recognition, 2008. ICPR 2008. 19th International Conference on*. IEEE, 1–4.
[20] Chris Seiffert, Taghi M Khoshgoftaar, Jason Van Hulse, and Amri Napolitano. 2010. RUSBoost: A hybrid approach to alleviating class imbalance. *IEEE Transactions on Systems, Man, and Cybernetics-Part A: Systems and Humans* 40, 1 (2010), 185–197.
[21] Liming Zhao and J Hayes. 2006. Predicting classes in need of refactoring: an application of static metrics. In *Proceedings of the 2nd International PROMISE Workshop, Philadelphia, Pennsylvania USA*.