An Empirical Study to Investigate Different SMOTE Data Sampling Techniques for Improving Software Refactoring Prediction

Rasmita Panigrahi[1](✉), Lov Kumar[2], and Sanjay Kumar Kuanar[1]

[1] GIET University, Gunupur, Odisha, India
{rasmita,sanjay.kuanar}@giet.edu
[2] BITS Pilani Hyderabad, Hyderabad, India
lovkumar505@gmail.com

Abstract. The exponential rise in software systems and allied applications has alarmed industries and professionals to ensure high quality with optimal reliability, maintainability etc. On contrary software companies focus on developing software solutions at the reduced cost corresponding to the customer demands. Thus, maintaining optimal software quality at reduced cost has always been the challenge for developers. On the other hand, inappropriate code design often leads aging, smells or bugs which can harm eventual intend of the software systems. However, identifying a smell signifier or structural attribute characterizing refactoring probability in software has been the challenge. To alleviate such problems, in this research code-metrics structural feature identification and Neural Network based refactoring prediction model is developed. Our proposed refactoring prediction system at first extracts a set of software code metrics from object-oriented software systems, which are then processed for feature selection method to choose an appropriate sample set of features using Wilcoxon rank test. Once obtaining the optimal set of code-metrics, a novel ANN classifier using 5 different hidden layers is implemented on 5 open source java projects with 3 data sampling techniques SMOTE, BLSMOTE, SVSMOTE to handle class imbalance problem. The performance of our proposed model achieves optimal classification accuracy, F-measure and then it has been shown through AUC graph as well as box-plot diagram.

Keywords: Software refactoring prediction · Code smell · Artificial Neural Network

1 Introduction

In the last few years, software has emerged as one of the most important form of technology to meet major decision-centric computational demands pertaining to business, Defence, communication, industrial computing and real-time control,

© Springer Nature Switzerland AG 2020
H. Yang et al. (Eds.): ICONIP 2020, CCIS 1332, pp. 23–31, 2020.
https://doi.org/10.1007/978-3-030-63820-7_3

security, healthcare, scientific research etc. Undeniably, the existence of modern human life can't be expected without software computing environment. The efficacy and unavoidable significance of software technologies have broadened the horizon for scientific, social as well as business communities to exploit it for optimal decision-making purposes. Being a significant need of modern socio-economic and scientific needs, software industry has taken a broadened shape inviting gigantically large-scale business communities to explore better technologies for better and enhanced productivity. Refactoring is the rework of existing code into well-designed code, and therefore assessing a code for its refactoring probability can be of utmost significance to ensure quality-software solution. Refactoring can help developers identifying bugs, improper design and vulnerability to strengthen the quality of the software product by means of enhanced logic-programme and complexity-free development. Though, authors have made different efforts to deal with refactoring problem such as analyzing structural elements, graphs, code metrics etc, identifying an optimal signifier has always remained a challenge. Recently, authors found that among the major possible solutions, exploiting software code-metrics can be vital to assist method-level refactoring proneness estimation. Refactoring can be defined as the modification of non-functional parameters without altering its desired output. Refactoring can be method level, class level, variable level, etc. Our work is all about the method level refactoring. We have considered seven different method level refactoring operations for our analysis such as: Extract Method, Inline Method, Move Method, Pull up Method, Push down Method, Rename Method and Extract and Move method.

In this paper a multi-purposive effort is made which intends to identify most suitable code-metrics and classification environment to perform method-level refactoring prediction or assessment. As a solution in this research a novel refactoring prediction model is developed for real-time software systems which obtains a set of source code metrics from software system by source meter tool. The obtained features are further processed to get the optimal code metrics by appropriate feature selection technique. After getting the significant features one of the statistical test will be conducted to select appropriate set of significant features (i.e. Wilcoxon test). This paper implements Artificial Neural Network for refactoring prediction at method level as well as to improve the prediction different SMOTE data sampling techniques are used with an empirical study.

- Q1: Whether the model gives any different results depending upon a different number of layers.
- Q2: which data sampling technique gives the optimal solution?
- Q3: All features or significant features give a good result.

2 Related Work

Martin Fowler has published a book "Refactoring: Improving the design of code" on1999. After its publication it has become the challenge for every researcher.

Earlier Mens and Tourwe [1] have done the survey on refactoring activities, tools support and supporting techniques. They have focused on the necessity of refactoring, code refactoring and design refactoring. Specifically, authors have shared their viewpoint on the impact of refactoring towards to software quality. Rosziati lbrahim et al. [2] has proposed a tool named as DART(Detection and refactoring tool) to detect the code smell and implement its corresponding refactoring activities without altering the system's functionality. Over the years empirical studies have recognized a correlation between code quality and refactoring operations. Kumar et al. [3] worked on a class-level refactoring prediction by applying machine learning algorithm named Least Squares Support Vector Machines (LSSVM) with different kernels and principal component analysis (PCA) as a feature extraction technique. To deal with data imbalance issue, authors applied synthetic minority over-sampling (SMOTE) technique. Employing different software metrics as refactoring-indicator authors performed refactoring prediction, where LSSVM with radial basis function (RBF) was found performing than the other state-of-art methods.

3 Study Design

This section presents the details regarding various design setting used for this research.

3.1 Experimental Data Set

There was a repository known as tera-PROMISE, which is publicly assessable by any researcher. The tera-promise repository contains open source projects related to software engineering, effort estimation, faults, source code analysis. Our data set has been downloaded from the tera-PROMISE repository, which makes our work easy. The tera-PROMISE repository is the standardized repository, which is manually validated by Kedar [4] and shared the data set publicly. We have taken five open source java projects which are present in GitHub Repository with subsequent releases.

3.2 Research Contribution

The presented work in this paper shows a novel and something different research contributions. In this paper, we are computing source code metrics at the method level. Basing up on the existing work, our study is on refactoring prediction at method level on 5 open source java projects (i.e. Antr4, titan, junit, mct, oryx) using Artificial Neural Network with 5 different hidden layers (ANN+1HL, ANN+2HL, ANN+3HL, ANN+4HL, ANN+5HL) and to improve the efficiency of software prediction different data sampling techniques (i.e. SMOTE, BLSMOTE, SVSMOTE).

4 Research Methodology

This section describes the model followed by an experiment implementing the Artificial Neural Network with 5 different hidden layers for refactoring prediction at method level with 3 different data sampling techniques. Figure 1 shows the outline of the proposed model for refactoring prediction at the method level by considering 5 open source java projects. Figure 2 identifies that the approach which we have proposed contains a multi step. 1st of all data set has to be collected from the tera-PROMISE repository. The source meter tool is implemented for source code metrics calculation. Significant features are to be selected through the Wilcoxon rank test, and Min-Max normalization is to carried for feature scaling, and then Data imbalance issues can be sorted through 3 data sampling techniques. ANN classifier is used for training the model, and lastly, the performance of the model is evaluated through different performance parameters (i.e., AUC, Accuracy, and F-measure). During the first phase, the data has to be pre-processed, where significant features are to be extracted by the Wilcoxon rank-sum test. Model building is the second phase, which consists of data normalization that may cause data balancing. Data unbalance issues can be solved by 3 data sampling techniques (i.e., SMOTE, BLSMOTE, and SVSMOTE).

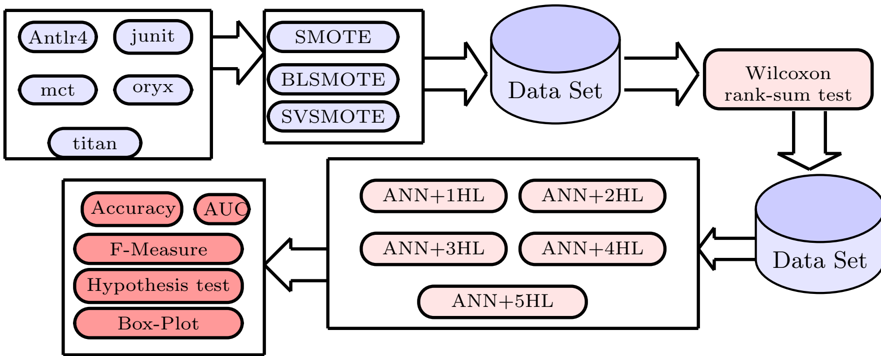


Fig. 1. Proposed model for refactoring prediction at the method level

5 Experimental Results

In this paper, we have used the ANN technique for refactoring prediction at the method level, and then it's performance has been improved by SVSMOTE data sampling technique and this has been represented in 3 performance measures (AUC, ACCURACY, F-measure). If we will talk about the performance in terms of ACCURACY of ANN gives likely same result irrespective of number of hidden layers. Like this, the performance of ANN in terms of AUC and F-measure gives the same type of result irrespective of a number of hidden layers. From the above table we can conclude that increasing or decreasing number

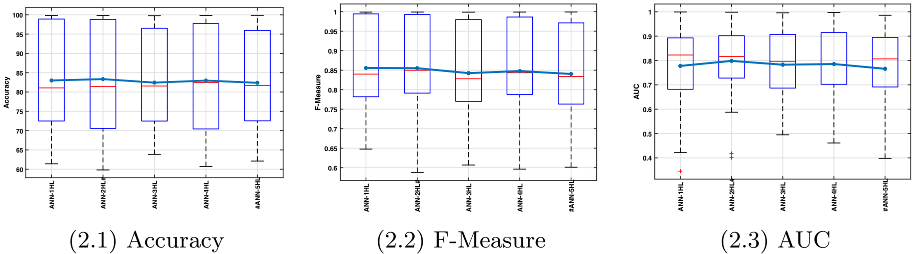
of hidden layers of ANN technique does not affect to its performance. Table 3 focuses on the ANN technique's performance in 3 different performance measures (ACCURACY, AUC) (Table 1).

Table 1. Performance Value: Classification Techniques

			Accuracy					AUC				
			1HL	2HL	3HL	4HL	5HL	1HL	2HL	3HL	4HL	5HL
ORG	AM	antlr4	98.85	98.47	98.85	98.73	98.6	0.95	0.91	0.95	0.98	0.7
ORG	AM	JUnit	99.38	99.43	99.38	99.47	99.43	0.5	0.59	0.53	0.65	0.43
ORG	SG	antlr4	98.51	98.66	98.57	98.7	98.73	0.66	0.74	0.69	0.62	0.64
ORG	SG	JUnit	99.03	99.38	99.38	99.29	99.38	0.66	0.74	0.6	0.46	0.52
SMOTE	AM	antlr4	99.71	99.31	86.28	90.13	79.88	1	1	0.95	0.96	0.89
SMOTE	AM	JUnit	81.03	81.18	81.27	80.67	77.79	0.89	0.9	0.88	0.89	0.86
SMOTE	SG	antlr4	77.05	79.03	78.38	80.73	81.06	0.85	0.86	0.87	0.88	0.89
SMOTE	SG	JUnit	65.45	67.42	66.78	66.33	71.82	0.78	0.79	0.78	0.77	0.81

5.1 Artificial Neural Network Classifier Results

In this paper we have focused on ANN classifier with 5 different hidden layers. In consideration of ANN classifier with its five hidden layers performance, we get all the hidden layers performance is likely the same depending upon different performance measures (AUC, Accuracy, F-measures). AUC performance of all the hidden layers of ANN is coming within a range .75 to .8, which is less than the mean AUC value. Accuracy performance of ANN with all hidden layers is in a range 80 to 85%, which is more than the mean accuracy. When we are considering F-measure performance, all are achieving more than .8, and that is more than their mean F-measure value. Figure 2 shows the performance of the ANN classifier in the form of the Box-plot diagram.

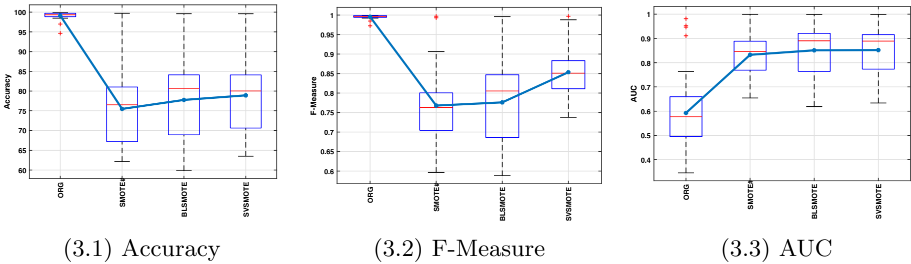


(2.1) Accuracy (2.2) F-Measure (2.3) AUC

Fig. 2. Performance evaluation of ANN with different Hidden layer

5.2 Data Imbalance Issue Results

Classifier building is very difficult if data is imbalanced. There is a technical test for the researcher to build an efficient model for refactoring prediction when data is unbalanced. This problem can be solved by an efficient technique i.e. Synthetic minority over sampling (SMOTE). SMOTE combines both the under-sampled as well as over-sampled. BLSMOTE is another over sampling technique which considers only boarder-line samples. SVSMOTE is one kind of over-sampling technique which uses SVM (support vector Machine) algorithm for the samples which is near to boarder-line. The performance of the sampling techniques (SMOTE, BLSMOTE, SVSMOTE) has been shown through the Box-plot diagram in Fig. 3. In the diagram we can see that SMOTE and SVSMOTE perform well that is more then its mean F-measure, But BLSMOTE gives 0.76 result which is less than mean F-measure performance. The performance result in terms of AUC of all the three imbalancing technique gives less than the mean AUC value. SVSMOTE gives the better performance the other SMOTE techniques. **Research answer Q2:-** From the experiment, we obtained that ORG means original data, and out of all three sampling techniques, SVSMOTE outperforms well.



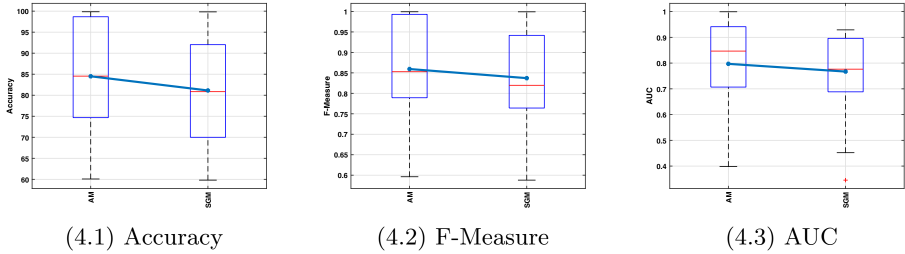
(3.1) Accuracy (3.2) F-Measure (3.3) AUC

Fig. 3. Performance evaluation of data imbalance issue

5.3 Significant Feature Results

During model construction feature selection is an important factor. Feature selection enables the machine learning algorithm to train the model faster. It reduces the model complexity and increases the model accuracy. It decreases the overfitting problem. Feature selection is useful due to its unlock capability for the potential uplifting of the model. Feature selection is used for dimensionality reduction to improve the accuracy and performance in a high dimensional data set. Feature importance plays a vital role during feature selection in a predictive modelling project. Feature importance means to class of techniques to assign the score to input featuresfor a predictive model. In this work we have focused two types of considerations with all features and significant features. We have got the experimental result that significant features gives well performance as compare

to all features. We have measured the model's performance in terms of AUC, Accuracy and F-measure in this article. During F-measure computation we have found that significant features and all features give the result, which is more than the mean F-measure value. Where as AUC computation all features gives the result of less than the mean AUC value. Both all features and significant features give same result as mean Accuracy. All the performances of all features and significant features have been shown on Fig. 4 in terms of box-plot diagram. **Research Answer Q3: -** Model Proposed gives good results with significant features as a comparison to all features.



(4.1) Accuracy (4.2) F-Measure (4.3) AUC

Fig. 4. Performance of all feature and significant features

5.4 Statistical Description of Significance Test Results

The learning efficiency, allied classification capacity makes ANN a potential solution for major Artificial Intelligence (AI) purposes or decision-making purposes. In this article, ANN, with its five different hidden layers, was used as a classifier. The statistical description of the performance of ANN with 5 different layers has been represented in Table 2. ANN with different layers does not effect on the model's performance. During the significance test of data sampling techniques, all the benefits in Table 2 are less than 0.5. It means a very less number of samples are imbalanced. So, it gives the optimal result for refactoring prediction at the method level. Table 3 provides the effect that significant features give a good result, so the model is accepted (Table 4).

Table 2. Statistical Significance test of ANN with different Hidden Layer

	ANN-1HL	ANN-2HL	ANN-3HL	ANN-4HL	ANN-5HL
ANN-1HL	1.00	0.61	0.96	0.88	0.72
ANN-2HL	0.61	1.00	0.68	0.77	0.42
ANN-3HL	0.96	0.68	1.00	0.88	0.72
ANN-4HL	0.88	0.77	0.88	1.00	0.62
ANN-5HL	0.72	0.42	0.72	0.62	1.00

Table 3. Statistical Significance test of SMOTE with its versions

	ORG	SMOTE	BLSMOTE	SVSMOTE
ORG	1.00	0.00	0.00	0.00
SMOTE	0.00	1.00	0.13	0.11
BLSMOTE	0.00	0.13	1.00	1.00
SVSMOTE	0.00	0.11	1.00	1.00

Table 4. Statistical Significance test of features

	AM	SGM
AM	1	0.027961344
SGM	0.027961344	1

6 Conclusion

In this paper, an explorative effort was made to identify an optimal computing environment for refactoring prediction purposes that, as a result, could help to achieve cost-efficient and reliable software design. In the proposed method, different code-metrics features, including object-oriented code metrics, were taken into consideration to characterize each code-class as refactoring prone or non-refactoring. With this motive obtaining a large number of code-metrics which describes different programming aspects or code-characteristics such as coupling, cohesion, complexity, depth, dependency, etc. A set of metrics were obtained, which were subsequently processed for significant feature selection. The proposed model intended to retain only significant features for eventual classification and to achieve a feature selection method was applied. On the other hand, realizing the data imbalance problem, three different sampling methods, including SMOTE, BLSMOTE, SVSMOTE in conjunction with original samples, provided sufficient training data for classification. This research introduced a classification algorithm, including ANN, with its five different hidden layers. Thus, as a contributing solution, this research recommends implementing ANN with any number of hidden layers that will give us the same type of results. ANN is used to achieve optimal and automatic refactoring prediction systems that, as a result, can help firms or developers to design software with better quality, reliability, and cost-efficiency.

References

1. Mens, T., Tourwé, T.: A survey of software refactoring. IEEE Trans. Softw. Eng. **30**(2), 126–139 (2004)
2. Ibrahim, R., Ahmed, M., Nayak, R., Jamel, S.: Reducing redundancy of test cases generation using code smell detection and refactoring. Journal of King Saud University-Computer and Information Sciences, **32**(3), pp. 367–374 2018

3. Kumar, L., Sureka, A.: 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), pp. 90–99. IEEE (2017)
4. Kádár, I., Hegedus, P., Ferenc, R., Gyimóthy, T.: A code refactoring dataset and its assessment regarding software maintainability. In: 2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER), **1**, pp. 599–603. IEEE (2016)