Indian Journal of Science and Technology, Vol 9(10), DOI: 10.17485/ijst/2016/v9i10/85110, March 2016
ISSN (Print) : 0974-6846
ISSN (Online) : 0974-5645

Neural Network based Refactoring Area Identification in Software System with Object Oriented Metrics

Jaspreet Kaur[1] and Satwinder Singh[2*]

[1]C. S. E. and I. T. Department B. B. S. B. E. C, Fatehgarh Sahib, Punjab – 140407, India; er.jaspreetkaurkohli@gmail.com
[2]Centre for Computer Science and Technology, Central university of Punjab, Bathinda, India; satwindercse@gmail.com

Abstract

Objectives of the Study

(a) To study previously designed models for identification of refactoring area in Object Oriented Software Systems.
(b) To design a general framework or model that helps to easily identify the software code smells for a good quality of coding.
(c) To identify the bad smells in the code with a design of neural network based model with the help of object-oriented metrics and further to predict the performance of the proposed model using various evaluation parameters of confusion matrix.

Analysis/Methods: In this study, two different versions of Rhino (1.7r1 and 1.7r2) were taken as dataset. Object –Oriented metrics were taken as input data and the probability factor (occurrence or non-occurrence of a bad smell as output. Presence of a bad smell was considered as 1 and 0 means absence of bad smell. If there was at least one bad smell present in the code in a class, it was marked as smelly class. The tool used to extract the databases for collected object –oriented metrics and bad smells of these Rhino versions is PTIDEJ. Further, the data was tested on neural networks for different epochs to predict their performance. **Findings: a) Bad Smell Analysis:** Twelve design smells were considered to detect the presence of bad smell in code. If there was at least one bad smell present in the code in a class, it was marked as smelly class. **b) Neural Network Model Table:** Weight and bias factor for various predictors were calculated for different epochs (500, 1000, and 2000). It shows the weights assigned from input layer to hidden layer and from hidden layer to output neurons layer. After the training, the weights were tested on various datasets. **C) Performance Tables and Graphs:** In this, the Neural network proposed model was trained using different number of epochs to examine if the number of epochs used in training has any impact on the results or not. Further, the results for the accuracy of these models were shown. **Novelty/Improvement:** When the data was highly trained then the results were better. When the data was trained with 500 epochs, it was suitable for only with-in company projects but when the data was more trained than the model was also appropriate for cross projects. It was seen that when the data was trained with 1000 and 2000 epochs, the results of the proposed model were improved.

Keywords: Artificial Neural Networks (ANN), Bad smells, Logistic Regression, Object Oriented Metrics, Refactoring, Software Maintainability

1. Introduction

Software needs to be changed with time and that change can be due to change in requirements, change of technology, etc. or in other words, we can say that system requires maintenance. Sometimes even a minor change in the software, can lead to degradation of the system, the code loses its originality and becomes complex. This problem is not solved by using better software development methods and tools as within the same time frame;

*Author for correspondence

we have to implement more new requirements making the software more complex again. In order to deal with a complexity like this, there is a technique that reduces the software complexity by incrementally improving the internal software quality. Restructuring is a research domain which addresses this problem. In terms of object-oriented software development, it is known as refactoring. A process of restructuring an existing computer code – changing the factoring – without changing its external behaviour is known as refactoring. This is a technique that keeps the software maintainable. It does not change the external interactions of a system, but merely improves the existing design, thus in turn improving the performance, correctness, and the maintainability, which makes the software easier to understand and modify.

Refactoring is a process which improves the internal structure of a program without affecting its external behavior[1]. The observable or external behavior of the software does not get changed by refactoring. 72 different refactoring are cataloged by Fowler. They range from localized changes such as EXTRACT LOCAL VARIABLE, to more global changes such as EXTRACT CLASS. To decide whether certain software needs refactoring or not there is a list of bad code smells[2]. Any symptom which indicates something wrong is referred to as a code smell. Whether the code should be refactored or the overall design should be re-examined is indicated by the code smell. Certain characteristics which can be rectified using refactoring are exhibited by a bad code.

As there is an increasing use of Object-oriented methods in the new software development, there is a growing need to document as well as improve the current practices in the Object-oriented design and development. Various models are used for predicting the performance of object-oriented metrics. Models are based on statistical relationship between the measure of quality and measure of software attributes. Previous studies have shown that use of neural networks is more advantageous as compared to logistic regression, as neural networks are more flexible. Studied[3] how code smells evolve over time and proposed the solution approach - a new group of code smells. In this study comparison of eight data sets was done to identify the proportion of functional defects and evolvability issues. The analysis was helpful in building tool support. It was found that people's relationship with the organization and code was an important factor in affecting the evaluation results.

To indicate where refactorings might be applicable by automatically detecting program invariants implemented[4] the Daikon tool. One invariant was that a certain parameter of a method is always constant, or is a function of the other parameters of a method. It requires dynamic analysis of the runtime behavior, which is the main problem of this approach. to infer the program invariants, the application needs to be executed. To this extent, the tool uses a representative set of test suites. However, it is impossible to guarantee that a test suite covers all possible runs of a program. To detect program parts that require refactoring is the identification of bad smells which is the most widespread approach. Have[5] made a prediction model that integrates ten object oriented metrics. They have used a statistical technique to establish a strong relationship between metrics and maintenance effort in the Object-Oriented [OO] systems.

In the same line[6] conducted a study that compares manual and automated refactoring, which was enabled by an algorithm. According to the results, more than half of the refactorings were performed manually. More than one third of the refactorings performed by developers are clustered in time. On an average, 30% of the performed refactorings do not reach the Version Control System.

Used[7] windows based GUI application to detect bad smells of object-oriented metrics. Metrics were used to identify the characteristics of bad smells "lazy class', "long method", "comment lines" and "large class". From the experimental results, it was observed that calculated metric values play a significant role to remove bad smells as the refactoring methods can be directly applied on the source code using calculated metric values. Therefore quality of the software is improved.

Used[8] Bayesian Inference to Predict Smelly classes Probability in Open source software. Bayesian inference graphs can represent decision for finding the smells present in software system. Study demonstrates a statistical technique for estimating the smelly classes for any piece of software and presents the relationship between smelly classes and object-oriented metrics. Two different types of methodologies were used to perform this experiment. This study contributes to all code smell prediction techniques by designing a Logistic regression model and using Bayesian inference graphs.

To identify bad smells and propose adequate refactorings [9] used object-oriented metrics. To propose move method/attribute and extract/inline class refactorings, they focus on use relations. The distance-based cohesion

metric was the key underlying concept. The degree to which methods and variables of a class belong together is measured by the distance- based cohesion metric. The use of object-oriented metrics is well-suited to detect places in the source code that are in need of refactoring especially in combination with software visualization. Find[10] out the threshold values against the bad smell for the Chidamber and Kemerer (CK) metrics at five different levels by using Logistic regression. To validate the study, two different versions of jfreechart were used. From the experimental result, it was observed that the CK metrics have threshold effects at various risk levels and some metrics have a useful threshold value at different levels to identify the bad smell. It was also concluded that threshold values play a significant role in improving the software quality because classes having more than threshold values will increase the testing efficiency.

2. Materials and Methods

To perform this research, two different versions of Rhino (1.7r1 and 1.7r2) were taken as dataset which was then run in background of eclipse. Then the database was extracted from eclipse workspace using tool. Further, the data was tested on neural networks for different epochs to predict their performance.

A JavaScript engine written fully in Java which is managed by the Mozilla Foundation as an open source software is Rhino. Object – Oriented metrics were taken as input data and the probability factor (occurrence or non-occurrence of a bad smell as output. The bad smells checked for includes: Antisingleton, Blob, Class Data Should Be Private, Complex class, Spaghetti Code, Swiss Army Knife, Lazy Class, Long Method, Long Parameter list, Message Chains, Refused Parent Request, Large class. Presence of a bad smell was considered as 1 and 0 means absence of bad smell. If there was at least one bad smell present in the code in a class, it was marked as smelly class.

2.1 Software Tool Used

The tool used to extract the databases for collected Object – Oriented metrics and bad smells of these Rhino versions is PTIDEJ[11].

Following Object-Oriented Metrics are used in this research:

The number of pairs of methods with no common attributes references is **LCOM**.

LCOM1 = 0 indicates a cohesive class.

Table 1. Collected Metrics

Metric	Description
LOC	Line of Code
WMC	Weighted Methods per Class
RFC	Response for a Class
LCOM1,LOCM2,LCOM5	Lack of cohesion in methods
CBO	Coupling Between Object classes
DIT	Depth of Inheritance Tree

LCOM2: LCOM2 is the percentage of methods excluding specific attributes being accessed in the class, averaged over all the attributes. LCOM2 indicates an undefined value and is displayed as zero, if the number of methods\attributes is zero.

LCOM5: As per Henderson-Sellers[12], cohesion is measured as per following properties:-

- One-to-many relationship where each method and every attribute of the class; known as "Perfect Cohesion", yields the measure as 0.
- One-to-one relationship where each method references only a single attribute yields the measure as 1.

LOC: Software metric using the number of lines in the text of the program's source code to measure size of the program is LOC.

RFC: Response for a Class. The number of different methods that can be executed when an object of that class receives a message[13] is measured.

DIT: Depth of Inheritance Tree. A measure of the inheritance levels from the object hierarchy top for each class[13] is the DIT.

CBO: Coupling between object classes. It represents the number of classes coupled to a given class. A count of the number of non-inheritance related couples with other classes is CBO for a class. Two things are coupled if and only if at least one of them acts upon the other[13].

WMC: Weighted methods per class. Simply the sum of the complexities of its methods[13] is a class's weighted methods per class WMC metric.

2.2 Model Based Upon Neural Networks

Many network architectures have been developed for various applications. Different types of networks are available in Mat lab such as feed forward backprop; feed forward time delay, perceptron, Hopfield etc. For predicting the performance of various object-oriented metrics,

Feed Forward Neural Network (FFNN) is used in this research. Feed forward networks include a series of layers. The first layer has a connection from the network input. All subsequent layers have connections from the previous layers. The network's output is produced by the final layer.

In Figure 1, eight neurons are used at input layer and 3 neurons at hidden layer. The 8 inputs are object-oriented metrics which are: - CBO, DIT, LCOM1, WMC, RFC, LOC, LCOM2, LCOM5. The connection between the i^{th} and j^{th} neuron is characterized by the weight coefficient w_{ij}, reflects the degree of importance of the given connection in the neural network is reflected by the weight coefficient. The output of a layer can be determined by the following equations.

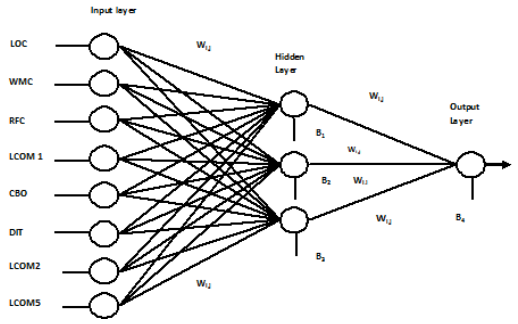

Figure 1. Neural Network Model.

$$a = x_1w_1 + x_2w_2 + x_3w_3 \ldots + x_nw_n$$

In this research, the following activation functions are used:-

- Hyperbolic Tangent Sigmoid Function (tansig)
- Linear Transfer Function (purelin)

Tansig is used as activation function for hidden layer and purelin is used as activation function for the output layer.Mean Square Error (MSE) is a network performance function which measures the network's performance according to the mean of squared errors. In this study, Gradient descent is used for updating the weights vector. Gradient descent method uses first-order derivative of total error to find the minima in error space.

2.2.1 Performance Evaluation Parameters

In this research, True Negative Rate (TNR), accuracy, precision and Mean Square Error (MSE) parameters are calculated to predict the performance of the proposed model using the confusion matrix of neural networks. A confusion matrix contains information about the actual and predicted classifications done by a classification system.

In Figure 2, the following are the four different possible outcomes of a single prediction for a two-class case with classes "1" ("yes") and "0" ("no").

- True Positives (TP)
- True negatives (TN)
- False Positives (FP)
- False Negatives (FN)

		Actual Class	
		Positives	Negatives
Predicted Class	Positives	True Positive (TP)	False positive (FP)
	Negatives	False Negative (FN)	True Negative (TN)

Figure 2. Confusion matrix of the proposed Model.

A false positive is when the outcome is incorrectly classified as "yes" (or "positive"), when it is in fact "no" (or "negative"). A false negative is when the outcome is incorrectly classified as negative when it is in fact positive. True positives and true negatives are correct classifications.

- TNR is the true negative rate that measures the proportion of negatives that are correctly identified. It is also called specificity.
- Precision is a description of a level of measurement that yields consistent results when repeated.
- Accuracy refers to the degree of conformity and correctness of something when compared to a true or absolute value. It also refers to the number of correct classifications divided by the total number of classifications.
- MSE measures a network's performance according to the mean of squared errors. It is a network performance function.

3. Results and Discussion

In this study, analysis is done by neural network based model on different versions of Rhino.

3.1 Bad Smell Analysis

Twelve design smells were considered to detect the presence of a bad smell in code**.** If there was at least one

bad smell present in the code in a class, it was marked as smelly class.

From the Table 2, it is clear that there is slight variation in results of both versions of Rhino. The affected classes are those classes which have bad smell. Number of affected classes is same in both Rhino Versions. From the Table 2, it is seen that in both RHINO versions, design smell which is found in most of the classes is BLOB. In rhino 1.7r1, antisingleton, large class, message chain and Spaghetti Code is the design smell, which is found in none of the class. In 1.7r2 large class, message chain, spaghetti code and Swiss Army Knife is the design smell which is found in none of the class.

Table 2. Analysis of Bad Smells

	Rhino 1.7r1	Rhino 1.7r2
Total Classes	62	61
Affected Classes	60	60
Antisingleton	0	1
Blob	27	27
Class Data Should Be Private	14	14
Complex Class	20	19
Large Class	0	0
Lazy Class	3	1
Long Method	19	17
Long Parameter List	11	11
Message Chains	0	0
Refused Parent Request	2	2
Spaghetti Code	0	0
Swiss Army Knife	1	0

3.2 Neural Network Model Table

In this, the Neural network proposed model was trained using different number of epochs to examine if the number of epochs used in training has any impact on the results or not. The results for the accuracy of these models are shown further.

In Tables 3 and 4, Rhino 1.7r1 and 1.7r2 were taken as a dataset for analysis. Weight and bias factor for various predictors are calculated for different epochs (500, 1000, and 2000). It shows the weights assigned from the input layer to the hidden layer and from the hidden layer to the output neurons layer. After the training, the weights were tested on various datasets

3.3 Performance Tables and Graphs

The following tables show the results of the proposed neural network model for 500, 1000, 2000 epochs. These datasets (1.7r1, 1.7r2) were trained using different iterations.

Table 3. Neural network model table of Rhino 1.7r1

		Epochs: 500				Epochs: 1000				Epochs: 2000			
		Hidden Layer 1			Output Layer	Hidden Layer 1			Output Layer	Hidden Layer 1			Output Layer
Predictor		H(1:1)	H(1:2)	H(1:3)	OUTPUT	H(1:1)	H(1:2)	H(1:3)	OUTPUT	H(1:1)	H(1:2)	H(1:3)	OUTPUT
Input Layer	(Bias)	1.6046	−0.05183	−1.419		1.8823	0.071111	−1.6089		−1.8209	0.17064	1.5941	
	CBO	−0.08993	0.23964	−0.84154		−0.25315	−0.49108	−0.0085105		0.63547	0.019521	0.062322	
	DIT	−0.34383	0.63866	0.29788		−0.79023	−0.61212	0.77808		0.61649	0.35438	0.76106	
	LCOM1	−0.69863	−0.77222	−0.86214		−0.11482	0.5331	0.44253		−0.34357	0.17484	0.13314	
	WMC	−0.09191	−0.17693	0.22703		−0.02412	0.3843	0.8481		−0.07251	−0.95385	0.66127	
	RFC	0.41636	0.048382	−0.66423		1.0485	−0.74559	0.81711		−0.2563	0.060044	0.59262	
	LOC	0.24628	−0.8283	−0.67483		0.048397	0.62809	0.064778		−0.83964	−0.82515	0.13772	
	LCOM2	−0.74851	−0.74197	−0.73753		−0.32549	0.35276	−0.60299		0.17458	0.53378	0.68976	
	LCOM5	−0.56782	−0.41473	−0.27206		0.023459	0.56418	0.16549		0.36415	−0.63456	−0.58322	
Hidden Layer 1	(Bias)				0.78416				−0.0789				0.4421
	H(1:1)				−0.1894				0.84795				−0.60404
	H(1:2)				−0.23795				−0.010775				0.28994
	H(1:3)				0.70159				−0.24057				0.36376

Table 4. Neural network model table of Rhino 1.7r2

Predictor		Epochs: 500				Epochs: 1000				Epochs: 2000			
		Hidden Layer 1			Output Layer	Hidden Layer 1			Output Layer	Hidden Layer 1			Output Layer
		H(1:1)	H(1:2)	H(1:3)	OUTPUT	H(1:1)	H(1:2)	H(1:3)	OUTPUT	H(1:1)	H(1:2)	H(1:3)	OUTPUT
Input Layer	(Bias)	1.534	0.063	1.607		1.603	0.033	−1.5756		−1.599	0.41	1.678	
	CBO	−0.912	−0.883	0.057		−1.065	−0.699	−0.57127		0.876	−0.51	0.516	
	DIT	0.671	0.784	−0.678		−0.427	−0.296	−0.80859		0.535	−0.434	0.066	
	LCOM1	0.176	−0.101	−0.89		0.1	−0.69	−0.60461		−0.907	−0.936	0.113	
	WMC	−0.368	−0.659	0.54		0.32	0.567	0.75166		0.584	0.968	−0.662	
	RFC	−0.431	0.062	−0.609		0.174	0.798	0.068329		0.138	0.053	−0.889	
	LOC	0.218	−0.467	0.283		0.04	−0.296	−0.13945		−0.354	−0.73	0.601	
	LCOM2	0.441	0.434	−0.088		−0.017	−0.73	0.60625		−0.407	0.026	−0.605	
	LCOM5	−0.908	−0.501	0.757		−1.057	−0.066	0.54695		0.211	−0.424	0.311	
Hidden Layer 1	(Bias)				1.085				0.983				−0.144
	H(1:1)				−0.66				−0.03				−0.052
	H(1:2)				0.465				0.38				0.837
	H(1:3)				0.167				0.456				0.253

Table 5. Performance table (500 epochs)

Training On	Testing On	%Accuracy	%TNR	Precision	MSE
Rhino 1.7r1	Rhino 1.7r2	94.9	0	96.6	0.0425
	Rhino 1.7r1	96.7	0	98.3	0.0259
Rhino 1.7r2	Rhino 1.7r2	96.8	0	96.8	0.0437
	Rhino 1.7r1	98.3	0	98.3	0.0368

The results of the proposed model for 500 epochs are shown above, in which the model was trained on one dataset and tested on other. It is observed that accuracy ranges from 94–99%. Rhino 1.7r1 has highest precision when it is tested over the same version but it doesn't give better accuracy results.

Related graphs are shown below:

In Figure 3 number of epochs is taken on x-axis and MSE on y-axis. The lower value of MSE, better the performance of the model. In Figure 3, when Rhino 1.7r1 is tested over same version, it gives low value of MSE; therefore it is better predictive model as compared to other ones.

In Table 6, best performance is given when training is done on 1.7r2 and testing on 1.7r1. Following Figure 4 shows the performance graph for Table 6.

As we can see in the figure, the graph of (d) is moving downwards because it has low value of MSE. The best training performance for (d) is 0.0184.

In Table 7, the highest accuracy is shown by Rhino 1.7r1 when it is tested over the same version and it has lower value of MSE i.e. the model performs better. The higher value of accuracy indicates that the model proposed in this research effectively discriminate refactoring area (bad smells) in the software, that is it has better predictive power.

Related Graphs are shown below:

From the Figure 5 1.7r1 (b), it is observed that best training performance is given by 1.7r1 when it is tested over the same version as compared to others. The graph is declining downwards more as compared to others for the same because it has low value of MSE. The results demonstrate that if the data is highly trained then the results will be better. When the data is trained with 500 epochs, it is suitable for only with-in company projects but when the data is more trained than the model is also appropriate for cross projects. As shown in Table 6 and 7 respectively, when the data is trained with 1000 and 2000 epochs, the results of the proposed model are improved.

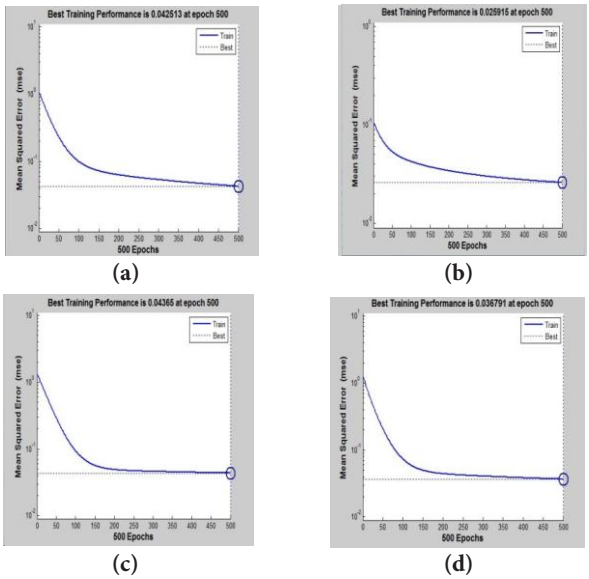

Figure 3. Training performance (MSE) for 500 epochs **(a)** 1.7r1 using 1.7r2, **(b)** 1.7r1 using 1.7r1, **(c)** 1.7r2 using 1.7r2, **(d)** 1.7r2 using 1.7r1.

Table 6. Performance table (1000 epochs)

Training On	Testing On	%Accuracy	%TNR	Precision	MSE
Rhino 1.7r1	Rhino 1.7r2	94.9	0	96.6	0.0356
	Rhino 1.7r1	96.7	0	98.3	0.0228
Rhino 1.7r2	Rhino 1.7r2	96.8	0	96.8	0.0349
	Rhino 1.7r1	98.3	0	98.3	0.0184

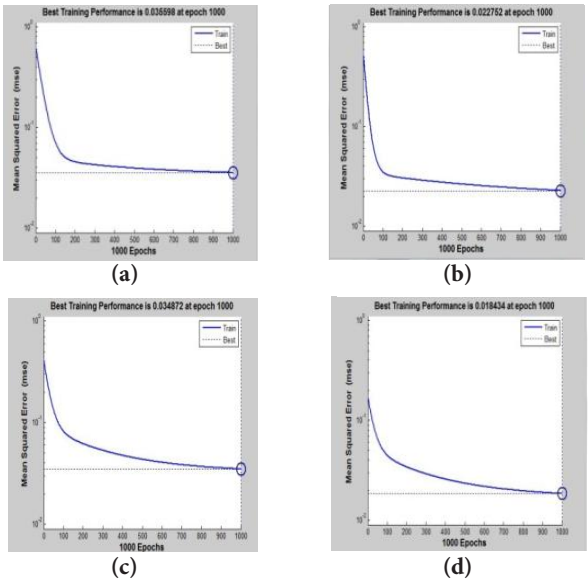

Figure 4. Training Performance (MSE) for 1000 epochs **(a)** 1.7r1using1.7r2, **(b)** 1.7r1using 1.7r1, **(c)** 1.7r2 using 1.7r2, **(d)** 1.7r2 using 1.7r1.

Table 7. Performance table (2000 epochs)

Training On	Testing On	%Accuracy	%TNR	Precision	MSE
Rhino 1.7r1	Rhino 1.7r2	94.9	0	96.6	0.0315
	Rhino 1.7r1	98.4	0	98.4	0.0175
Rhino 1.7r2	Rhino 1.7r2	95.2	0	96.7	0.0340
	Rhino 1.7r1	98.3	0	98.3	0.0179

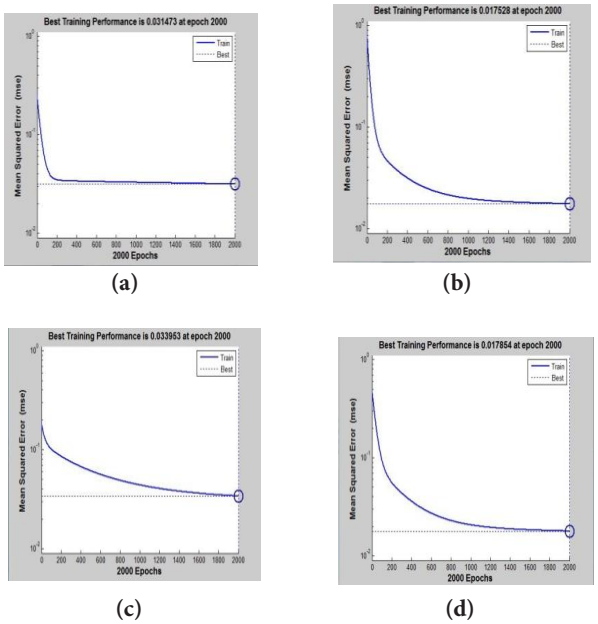

Figure 5. Training Performance (MSE) for 2000 epochs **(a)** 1.7r1using1.7r2, **(b)** 1.7r1using 1.7r1, **(c)** 1.7r2 using 1.7r2, (d) 1.7r2 using 1.7r1.

4. Conclusion

In this study, two different rhino versions were validated using designed neural network model. Results varied when the performance was evaluated on different versions, for different epochs using same neural network model. Neural networks played a significant role in identifying the refactoring area with Object-Oriented metrics. This study also identifies the relationship between bad smells and object-oriented metrics. A variety of software tools have been developed for the automated detection of bad smells but they differ in their capabilities and approaches. The degree of automation level varies from tool to tool. Further studies should be done to investigate why certain refactoring tools are underused. Researchers should also consider how this knowledge can be used to rethink these tools. In general, there is a need for formalisms, processes, methods and tools which address refactoring in a more consistent, directed,

scalable and flexible way. It is found that code smells affects the quality of software system. Therefore good prediction system is necessary for predicting the performance of model. Our Proposed model produced better results even, when it was used over inter-project datasets in case of 1000 and 2000 epochs. These models can be further modified for cross-company projects for the better results.

5. Acknowledgement

I would like thank the Punjab Technical University, Jalandhar for giving me the opportunity to work on my thesis during my final year of M.Tech. Thesis as a research tool is an important aspect in engineering.

I also owe my sincerest gratitude towards Dr. Satwinder Singh (Asstt. Prof. Centre for Computer Science and Technology, Central university of Punjab, Bathinda.) for his valuable advice, continuous guidance, and a balanced criticism throughout my thesis which has helped me immeasurably to complete my work successfully.

6. References

1. Fowler M, Beck K, Brant J, Opdyke W, Roberts D. Refactoring: Improving the Design of Existing Code. Addison-Wesley; 1999.
2. Fowler M, Beck K. "Bad Smells in Code," in Refactoring: Improving the Design of Existing Code. 1st ed., Boston: Addison-Wesley; 2000. p. 75–88.
3. Mantyla M, Vanhanen J, Lassenius C. A Taxonomy and Initial Empirical Study of Bad Smells in Code. Proceedings of the International Conference on Software Maintenance; 2003. p. 381.
4. Kataoka Y, Ernst M D, Griswold W G, Notkin D, "Automated Support for Program Refactoring Using Invariants," Proc. International Conference on Software Maintenance, 2001, pp. 736- 43.
5. Li W, Henry S. Object-Oriented Metrics that Predict Maintainability. Journal of Systems and Software. 1993; 23(2):111–22.
6. Negara S, Chen N, Vakilian M, Johnson RE, Dig D. A Comparative Study of Manual and Automated Refactorings. European Conference on Object-Oriented Programming (ECOOP); 2013. p. 552–76
7. Rani A, Kaur H. Detection of Bad Smells in Source Code According To Their Object Oriented Metrics. International Journal for Technological Research in Engineering. 2014; 1(10): 1211–14.
8. Kapila H, Singh S. Bayesian Inference to Predict Smelly classes Probability in Open source software. International Journal of Current Engineering and Technology. 2014 June; 4(3): 1724–28.
9. Steinbruckner SF, Lewerentz C.Metrics Based Refactoring. Proc. European Conf. Software Maintenance and Reeng; 2001. p. 30–38.
10. Kaur S, Singh S, Kaur H. A Quantitative Investigation of Software Metrics Threshold Values at Acceptable Risk Level. International Journal of Engineering Research and Technology (IJERT). 2013 March; 2(3):1–7.
11. Gueheneuc YG. A Reverse Engineering Tool for Precise Class Diagrams. CASCON; 2004.
12. Jamali SM. Object Oriented Metrics (A Survey Approach). 2006.
13. Chidamber SR and Kemerer CF. A metrics suite for object oriented design. IEEE Trans. On Software Engineering. 1994; 20(6):476–493.
14. Satwinder S, Kahlon KS. Effectiveness of encapsulation and Object-Oriented –Metrics to refactor code and identify error prone classes using bad smells. ACM SIGSOFT Software Engineering Notes. 2011; 36(5): 1–10.
15. Satwinder S, Kahlon KS. Effectiveness of refactoring metrics model to identify smelly and error prone classes in open source software. ACM SIGSOFT Software Engineering Notes. 2012; 37(2): 1–11.
16. Satwinder S, Kahlon KS. Object-Oriented metrics threshold values at quantitative acceptable risk level. CSI transactions on ICT; 2014; 2(3):191–205.
17. Satwinder S, Mittal P, Kahlon KS. Empirical model for predicting high, medium and low severity faults using object-oriented metrics in Mozilla Firefox. International Journal of Computer Applications in Technology. 2013; 47(2/3): 110–124.