International Journal of Computer Applications Technology and Research
Volume 5– Issue 5, 260 - 265, 2016, ISSN:- 2319–8656

Software Defect Prediction Using Radial Basis and Probabilistic Neural Networks

Riyadh A.K. Mehdi
College of Information Technology
Ajman University of Science and Technology
Ajman, United Arab Emirates

Abstract: Defects in modules of software systems is a major problem in software development. There are a variety of data mining techniques used to predict software defects such as regression, association rules, clustering, and classification. This paper is concerned with classification based software defect prediction. This paper investigates the effectiveness of using a radial basis function neural network and a probabilistic neural network on prediction accuracy and defect prediction. The conclusions to be drawn from this work is that the neural networks used in here provide an acceptable level of accuracy but a poor defect prediction ability. Probabilistic neural networks perform consistently better with respect to the two performance measures used across all datasets. It may be advisable to use a range of software defect prediction models to complement each other rather than relying on a single technique.

Keywords: Software defect prediction; datasets; neural networks; radial basis functions; probabilistic neural networks.

1. INTRODUCTION

Defects in modules of software systems is a major problem in software development. Software failure of an executable product or non-conformance of software to its functional requirements is a software defect. A software defect is a fault, error, or failure in a software module that results in incorrect result or unexpected behavior. These defects can arise from mistakes and errors made during a software coding or in its design and few are caused by compilers producing incorrect code. Predicting faulty software modules and identifying general software areas where defects are likely to occur could help in planning, controlling and executing software development activities and save time and money [1]. In the context of software engineering, software quality refers to software functional quality and software architectural quality. Software functional quality reflects functional requirements whereas architectural quality emphasizes non-functional requirements. The objective of software product quality engineering is to achieve the required quality of the product through the definition of quality requirements and their implementation, measurement of appropriate quality attributes and evaluation of the resulting quality [2].

A software metric is a standard of a quantitative measure of the degree to which a software system or process possesses some property. Metrics are functions, while measurements are the numbers obtained by the application of metrics. Metrics allow us to gain understanding of relationships among processes and products and build models of these relationships. Software quality metrics are a subset of software metrics that focus on the quality aspects of the product, process, and project. Product metrics describe the characteristics of the product such as size, complexity, design features, performance, and quality level. Process metrics can be used to improve software development and maintenance such as the effectiveness of defect removal during development, the pattern of testing defects arrival, and the response time of the fix process. Project metrics describe the project characteristics and execution which includes the number of software developers, the staffing pattern over the life cycle of the software, cost, schedule, and productivity [3][4]. Software defect prediction refers to those models that try to predict potential software defects from test data. There exists a correlation between the software metrics and the existence of a fault in the software. A software defect prediction model consists of independent variables (software metrics) collected and measured during software development life cycle and a dependent variable (defective or non-defective software) [2].

2. LITERATURE REVIEW

Wahono [4] provides a systematic literature review of software defect prediction including research trends, datasets, methods and frameworks. There are a variety of data mining techniques used to predict software defects such as regression, association rules, clustering, and classification. This paper is concerned with classification based software defect prediction. Various classification techniques have been used such as:

- Neural Networks
- Decision trees
- Naïve Bayes
- Support Vector Machines
- Case Based Reasoning

In their work, Okutan and Yildiz [5] used a Bayesian networks to determine the set of metrics that are most important and focus on them more to predict defectiveness. They used the Bayesian networks to determine the probabilistic influential relationships among software metrics and defect proneness. In addition to the metrics used in Promise data repository, they defined two more metrics, i.e. NOD for the number of developers and LOCQ for the lack of code quality. They extract these metrics by inspecting the source code repositories of the selected Promise data repository data sets. From the model they can determine the marginal defect proneness probability of the whole software system, the set of most effective metrics, and the influential relationships among metrics and defectiveness. Their experiments on nine open source Promise data repository data sets show that response for class (RFC), lines of code (LOC), and lack of coding quality (LOCQ) are the most effective metrics whereas coupling between objects (CBO), weighted method per class (WMC), and lack of cohesion of methods (LCOM) are less effective metrics on defect proneness. Furthermore, number of children (NOC) and depth

of inheritance tree (DIT) have very limited effect and are untrustworthy. On the other hand, based on the experiments on Poi, Tomcat, and Xalan data sets, they observed that there is a positive correlation between the number of developers (NOD) and the level of defectiveness. However, they stated that further investigation involving a greater number of projects is needed to confirm their findings.

Kaur and Kaur [6] have tried to find the quality of the software product based on identifying the defects in the classes. They have done this by using different classifiers such as Naive base, Logistic regression, Instance based (Nearest-Neighbour), Bagging, J48, Decision Tree, Random Forest. This model is applied on five different open source software to find the defects of 5885classes based on object oriented metrics. Out of which they found only Bagging and J48 to be the best.

Li and others [7] described three methods for selecting a sample: random sampling with conventional machine learners, random sampling with a semi-supervised learner and active sampling with active semi-supervised learner. To facilitate the active sampling, we propose a novel active semi-supervised learning method ACoForest which is able to sample the modules that are most helpful for learning a good prediction model. Our experiments on PROMISE datasets show that the proposed methods are effective and have potential to be applied to industrial practice.

Gao and Khoshgoftarr [8], presented an approach for using feature selection and data sampling together to deal with the problems. Three scenarios are considered: 1) feature selection based on sampled data, and modeling based on original data; 2) feature selection based on sampled data, and modeling based on sampled data; and 3) feature selection based on original data, and modeling based on sampled data. Several software measurement data sets, obtained from the PROMISE repository, are used in the case study. The empirical results demonstrated that classification models built in scenario 1) result in significantly better performance than the models built in the other two scenarios. In their work, Purswani and others [9] have combined a K-means clustering based approach with a feed-forward neural network using PC1 data set from NASA MDP software projects. The performance was based on MAE and RMSE values. Results have shown that this hybrid approach is better than analytical approaches.

Artificial Neural Networks (ANN) have been used in software defect prediction. ANNs are inspired by the way biological nervous system works, such as brain processes an information. An ANN mimics models of biological system, which uses numeric and associative processing. In two aspects, it resembles the human brain. First it acquires knowledge from its environment through a learning process. Second, synaptic weights that are used to store the acquired knowledge, which is interneuron connection strength. There are three classes of neural networks, namely single layer, multilayer feed forward networks and recurrent networks. Neural networks have been shown to perform well in classification tasks. However, there are different neural networks architectures. This paper investigates the effectiveness of using a radial basis function neural network and a probabilistic neural network on prediction accuracy and defect prediction compared with other approaches [10].

3. RESEARCH METHODOLOGY

The most commonly used neural network architecture is the back propagation trained multilayered feed forward networks with sigmoidal activation function [11]. Each neuron in a MLP takes the weighted some of its input values. That is, each input value is multiplied by a coefficient, and the results are all summed together. A single MLP neuron is a simple linear classifier, but complex non-linear classifiers can be built by combining these neurons into a network.

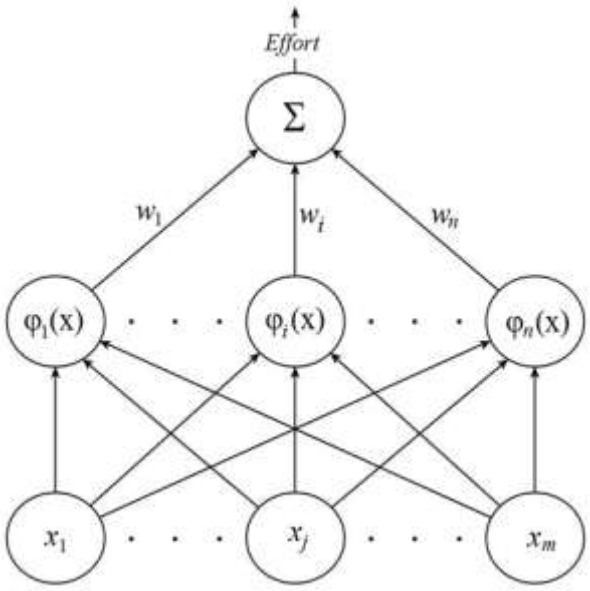

Figure 1. Radial Basis Function Neural Network

3.1 Radial Basis Function Neural Network

Another type of ANN that has been used in the literature is the radial basis functions neural network (RBFNN) [12]. A RBFNN can approximate any function which makes it suitable to model the relationship between inputs (the various cost drivers) and output (effort required). RBFNN performs classification by measuring the input's similarity to examples from the training set. Each RBFNN neuron stores a "prototype", which is just one of the examples from the training set. To classify a new input, each neuron computes the Euclidean distance between the input and its prototype. In other words, if the input more closely resembles class A prototypes than the class B prototypes, it is classified as class A. It has been show that RBFNN perform better than other types of neural networks based models [13]. In this paper we will use RBFNN to examine the effect of preprocessing datasets with PCA on the accuracy of software effort estimation models.

3.2 RBFNN Implementation

The following generic description of a RBF neural network is based on a tutorial given in [11][12]. Figure 1 describes a typical architecture of an RBFNN. It consists of an input vector, a layer of RBF neurons, and an output layer with one node per category or class of data. The input vector is the m-dimensional vector to be classified. The hidden layer consists of neurons where each one stores a "prototype" vector which is just one of the vectors from the training set. Each RBFNN neuron compares the input vector to its prototype, and outputs a value between 0 and 1 which is a measure of similarity. If the input is equal to the prototype, then the output of that RBFNN neuron will be 1. As the distance between the input and prototype grows, the response falls off exponentially towards zero.

The neuron's response value is also called its "activation" value. The prototype vector is also often called the neuron's

"center", since it's the value at the center of the bell curve. The output of the network consists of a set of nodes, one per category to be classified or a value to be computed. Each output node computes a sort of score for the associated category. Typically, a classification decision is made by assigning the input to the category with the highest score. The score is computed by taking a weighted sum of the activation values from every RBF neuron. Because each output node is computing the score for a different category, every output node has its own set of weights. The output node will typically give a positive weight to the RBF neurons that belong to its category, and a negative weight to the others.

Each RBFNN neuron computes a measure of the similarity between the input and its prototype vector (taken from the training set). Input vectors which are more similar to the prototype return a result closer to 1. There are different possible choices of similarity functions, but the most popular is based on the Gaussian function. Equation 1 describe the formula for a Gaussian with a one-dimensional input.

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}} \quad \text{--------- (1)}$$

Where x is the input, μ is the mean, and σ is the standard deviation. This produces the familiar bell curve shown below in Figure 2, which is centered at the mean, μ. In the Gaussian distribution, μ refers to the mean of the distribution. Here, it is the prototype vector which is at the center of the bell curve.

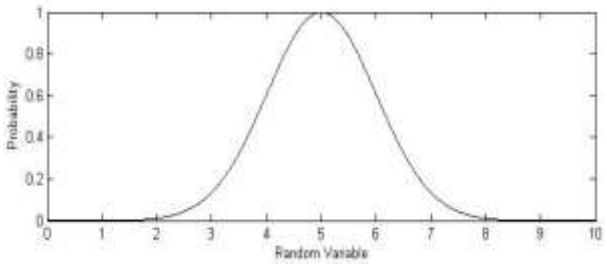

Figure 2. A Gaussian function with μ=5, and σ=1.

For the activation function, $\varphi(\ )$, the standard deviation, σ, is not important and the following two simplifying modifications can be made. The first modification is to remove the outer coefficient, 1 / (σ * sqrt (2 * π)). This term normally controls the height of the Gaussian curve. Here, though, it is redundant with the weights applied by the output nodes. During training, the output nodes will *learn* the correct coefficient or "weight" to apply to the neuron's response. The second change is to replace the inner coefficient, 1 / (2 * σ^2), with a single parameter β which controls the width of the bell curve. Again, in this context, the value of σ is not in itself important and what is needed is some coefficient that controls the width of the bell curve. Thus, after making these two modifications, the RBFNN neuron activation function can be written as in equation 2 [12]:

$$\varphi(x) = e^{-\beta\|x-\mu\|^2} \quad \text{--------- (2)}$$

There is also a slight change in notation here when we apply the equation to n-dimensional vectors. The double bar notation in the activation equation indicates that we are taking the Euclidean distance between x and μ, and squaring the result. For a 1-dimensional Gaussian, this simplifies to just $(x - \mu)^2$.

It is important to note that the underlying metric here for evaluating the similarity between an input vector and a prototype is the Euclidean distance between the two vectors. Also, each RBF neuron will produce its largest response when the input is equal to the prototype vector. This allows to take it as a measure of similarity, and sum the results from all of the RBFNN neurons. As we move out from the prototype vector, the response falls off exponentially. Every RBF neuron in the network will have some influence over the classification decision. The exponential fall off of the activation function, however, means that the neurons whose prototypes are far from the input vector will actually contribute very little to the result [12].

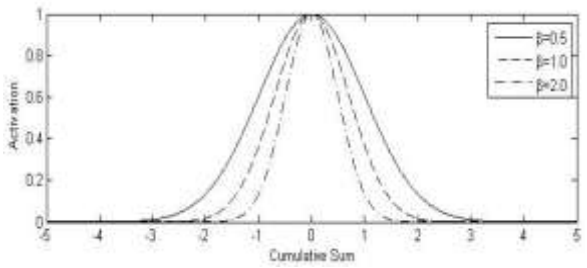

Figure 3. Activation Function for different values of beta.

3.3 Probabilistic Neural Networks

Probabilistic neural networks (PNN) can be used for classification problems. The following description of probabilistic neural networks is taken from the Neural Networks Toolbox documentation [11]. When an input is presented, the first layer computes distances from the input vector to the training input vectors, and produces a vector whose elements indicate how close the input is to a training input. The second layer sums these contributions for each class of inputs to produce as its net output a vector of probabilities. Finally, a *compete* transfer function on the output of the second layer picks the maximum of these probabilities, and produces a one for that class and a 0 for the other classes. The architecture for this system is shown in Figure 4.

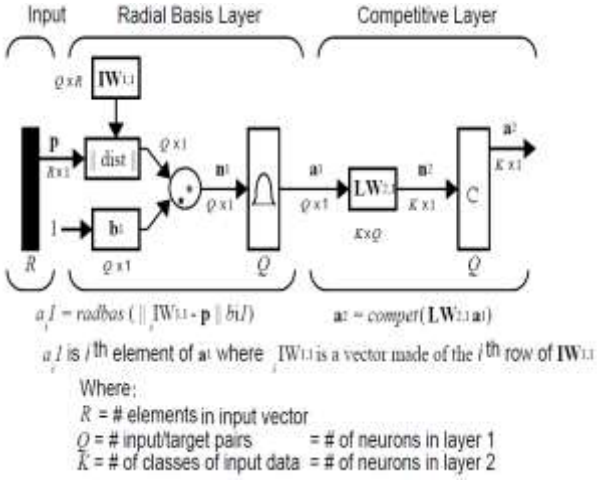

Figure 4. Probabilistic Neural Network Architecture.

It is assumed that there are Q input vector/target vector pairs. Each target vector has K elements. One of these element is one and the rest is zero. Thus, each input vector is associated with one of K classes. The first layer input weights, IW1,1 are set to

the transpose of the matrix formed from the Q training pairs, **P**'. When an input is presented the ||dist|| box produces a vector whose elements indicate how close the input is to the vectors of the training set. These elements are multiplied, element by element, by the bias and sent the *radbas* transfer function. An input vector close to a training vector will be represented by a number close to one in the output vector **a**1. If an input is close to several training vectors of a single class, it will be represented by several elements of **a**1 that are close to one. The second layer weights, LW1,2, are set to the matrix **T** of target vectors. Each vector has a one only in the row associated with that particular class of input, and zeros elsewhere. The multiplication **Ta**1 sums the elements of **a**1 due to each of the K input classes. Finally, the second layer transfer function, compete, produces a one corresponding to the largest element of **n**2, and zeros elsewhere. Thus, the network has classified the input vector into a specific one of K classes because that class had the maximum probability of being correct.

3.4 Software Defect Prediction Datasets

The following NASA software defect prediction datasets available publicly from the PROMISE repository are used in this research:

- KC1: a C++ system implementing storage management for receiving and processing ground data.
- KC2: same as KC1 but with different personnel.
- CM1: is a NASA spacecraft instrument written in "C".
- JM1: A real-time predictive ground system written in "C".
- PC1: is s flight software for earth orbiting satellite written in "C".

Defect detectors are assessed according based on the following confusion matrix:

		Modules actually has defects	
		No	Yes
Classifier predicts no defects	No	*a*	*b*
Classifier predicts some defects	Yes	*c*	*d*

Measures used based on the above confusion matrix are:

- *Acc*, Accuracy = $(a + d) / (a + b + c + d)$
- *PD*, Probability of detection $= d / (b + d)$

4. IMPLEMENTATION and RESULTS

MATLAB neural network toolbox [11] was used to implement the RBFNN and the PNN. For RBFNN, the *newrb* function is used:

Ne t= newrb (*P, T, goal, spread, MN, DF*), where

- *P* is a *m*×*n* matrix of input vector, *m* is the number of cost drivers for each project, and *n* is the number of projects used in the training phase.
- *T* is 1×*n* vector of actual efforts for each project used in the training phase.
- *Goal: mean squared errors*, 0.0001 is used.
- *Spread*: a value between 1.0 and 3.0 is used.
- *MN*: maximum number of neurons (default *n*)
- *DF*: number of neurons to add between displays, 2 is used.

For the PNN, the *newppnn* function is used:

Ne t= newpnn (*P, T, Spread*), where

- *P* is a *m*×*n* matrix of input vector, *m* is the number of attributes, and *n* is the number of input vectors used in the training phase.
- *T* is 1×*n* vector of target vectors used in the training phase.
- *Spread:* spread of radial basis functions (default = 0.1). If spread is near zero, the network acts as a nearest neighbor classifier. As spread becomes larger, the designed network takes into account several nearby design vectors.

Table 1 shows the results obtained from applying a radial basis and a probabilistic neural networks to the PROMISE datasets.

Table 1. Performance measures of applying RBFNN and PNN to the PROMISE datasets.

	RBFNN		PNN	
Dataset	*Acc*	*PD*	*Acc*	*PD*
KC1	%77.0	%44	%83.1	%29.7
KC2	%78.0	%50	%83.4	%50.0
CM1	%82.5	%20	%87.3	%33.3
JM1	%77.0	%31	%77.5	%30.0
PC1	%89.2	%32	%91.6	%40.0

Figure 5 shows the accuracy and prediction power of radial basis function neural net when applied to the PROMISE datasets. It can been seen that accuracy obtained is very close for the various datasets except for the PC1 dataset. As these data sets use the same software attributes and collected by the same processes, the difference in performance may be attributed to the nature of the PC1 project. The performance with respect to prediction of defects is not encouraging. This issue need to be investigated further.

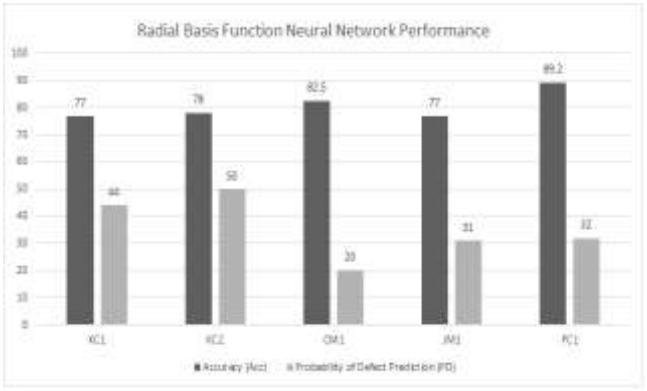

Figure 5. Accuracy and prediction power of RBF neural network to the PROMISE datasets.

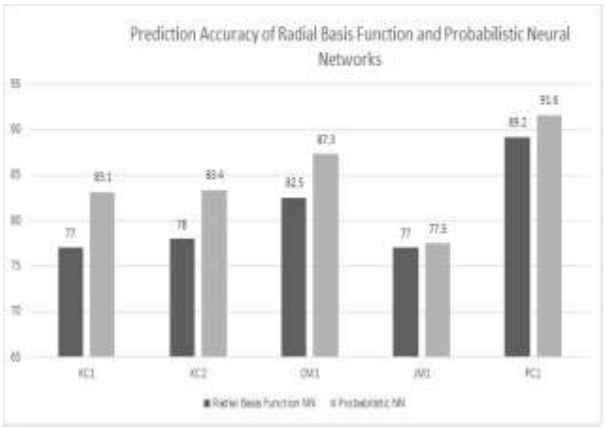

Figure 6. Actual, estimated, and PCA estimated efforts for the Cocomo81 data set.

The accuracy and defect prediction ability of probabilistic neural networks are depicted in Figure 6. Accuracy obtained is slightly better than that obtained from RBFNN. However, prediction ability is poor. However, it can be seen from Figures 7 and 8 that probabilistic neural nets performed consistently better that radial basis function neural networks with respect to accuracy and defect prediction ability.

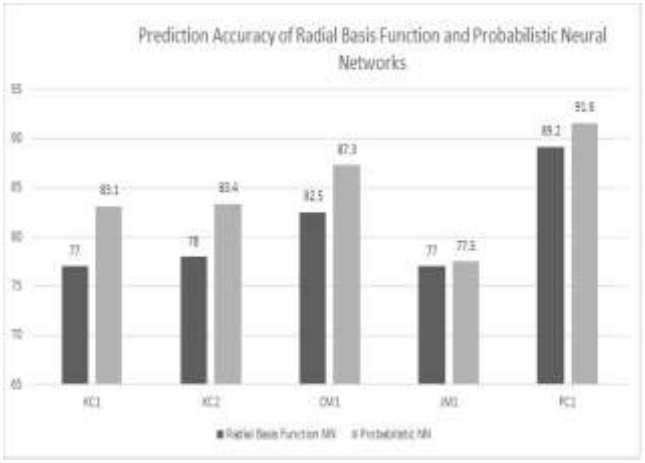

Figure 7. Actual, estimated, and PCA estimated efforts for the Maxwell data set.

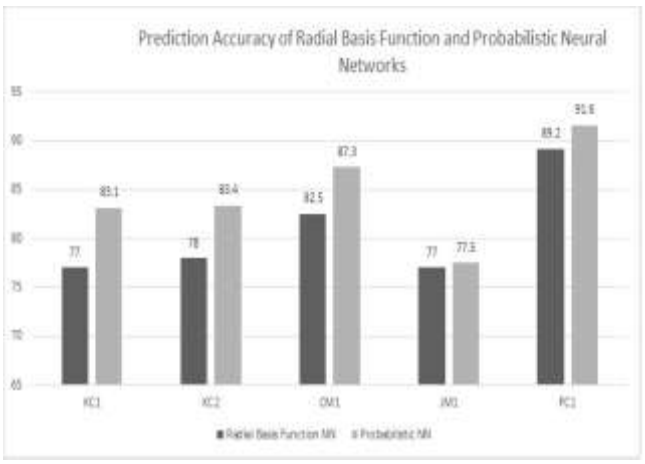

Figure 8. Actual, estimated, and PCA estimated efforts for the China data set.

5. CONCLUSIONS

The conclusions to be drawn from this work is that the neural networks used in here provide an acceptable level of accuracy but a poor defect prediction ability. Probabilistic neural networks perform consistently better with respect to the two performance measures used across all datasets. It may be advisable to use a range of software defect prediction models to complement each other rather than relying on a single technique. Further investigation of the use of neural network approached in software defect prediction is necessary to reach a solid conclusion.

6. ACKNOWLEDGMENTS

I would like to thank Ajman University of Science and Technology for providing the time and resources to conduct this research.

7. REFERENCES

[1]. K. Gupta, S. Kang, "Fuzzy Clustering Based approach for Prediction of Level of Severity of Faults in Software Systems," *International Journal of Computer and Electrical Engineering*, vol. 3, no. 6, 2011.

[2]. M. Prasad, L. Florence, and A. Arya, "A Study on Software Metrics Based Software Defect Prediction using Data Mining and Machine Learning Techniques," *International Journal of Database Theory and Application*, vol. 8, no. 3, 2015.

[3]. L. Madeyski, M. Jureczko, "Which process metrics can significantly improve defect prediction models? An empirical study," *Software Quality Journal*, vol. 23, issue 3, 2015.

[4]. R. S. Wahono, "A Systematic Literature Review of Software Defect Prediction: Research Trends, Datasets, Methods and Frameworks," Journal of Software Engineering, vol. 1, no. 1, 2015.

[5]. A. Okutan, O. T. Yildiz, "Software defect prediction using Bayesian networks," *Empirical Software Engineering*, vol. 19, no. 1, 2014.

[6]. A. Kaur, I. Kaur, "Empirical Evaluation of Machine Learning Algorithms for Fault Prediction," *Lecture Notes on Software Engineering*, vol. 2, no. 2, 2014.

[7]. M. Li, H. Zhang, R. Wu, and Z. H. Zho, "Sample-based software defect prediction with active and semi-supervised learning," *Automated Software Engineering*, vol. 19, no.2, 2012.

[8]. K. Gao, T. M. Khoshgoftaar, "Software Defect Prediction for High-Dimensional and Class-Imbalanced Data," 23rd International Conference on Software Engineering & Knowledge Engineering, Eden Roc Renaissance, Miami Beach, USA, 2012.

[9]. K. Purswani, P. Dalal, A. Panwar, and K. Dashora, "Software Fault Prediction Using Fuzzy C-Means Clustering and Feed Forward Neural Network,"

International Journal of Digital Application & Contemporary research, vol. 2, issue 1, 2013.

[10]. S. Haykin. *Neural Networks: A Comprehensive Foundation* (2nd Edition). Prentice-Hall International, 2013, pp. 43-45.

[11]. Neural Networks Toobox: User's Guide, The Mathworks, Inc., Natic, MA 01760-2098, 1992-2002.

[12]. Liu, *Radial Basis Function (RBF) Neural Network Control for Mechanical Systems.* Springer, 2013.

[13]. E. Praynlin, and P. Latha, "Performance Analysis of Software Effort Estimation Models Using Radial Basis Function Network," *International Journal of Computer, Information,* Systems and Control Engineering, vol. 8, no. 1, 2014.