International Journal of Engineering Applied Sciences and Technology, 2016
Vol. 2, Issue 1, ISSN No. 2455-2143, Pages 86-92
Published Online November-December 2016 in IJEAST (http://www.ijeast.com)

SOFTWARE DEFECT PREDICTION USING LEARNING to RANK APPROACH and BOLTZMANN LEARNING APPROACH

Bhanu Priya
Dept. of computer science and engineering
CTITR Jalandhar, India

Sarabjit Kaur
Assistant Professor
Dept. of computer science and engineering
CTITR
Jalandhar, India

***Abstarct*-** **The software engineering is the technology to process the software and perform various operations on that software . The testing the important application of software engineering in which test cases are applied to detect defects from the software . In the recent times, it is been analyzed that defects may also raised in the test cases which are used for the defect prediction. In this work, Rank-to-learn algorithm is applied for the prediction of defects from the software. To improve performance of Rank-to-learn algorithm in terms of defect prediction rate the technique of back propagation is applied which learn from the precious experience and drive new values. The system is tested on 10 test cases and simulation is performed in MATLAB. The simulation results show that the defect prediction rate is increased and execution time is reduced.**

Keywords -Defects, test Cases, neural networks, Boltzmann learning, learning to rank approach

I. INTRODUCTION

Software defects can lead to undesired results. To predict defective files, a prediction model must be built with predictors (e.g., software metrics) obtained from either a project itself (within-project) or from other projects (cross-project). A universal defect prediction model that is built from a large set of diverse projects would relieve the need to build and tailor prediction models for an individual project. The current software defects prediction mainly uses the software metrics to predict the amount and distribution of the software defects. The research method of software defects classification prediction is based on the program properties of the historical software versions, to build different prediction models and to forecast the defects in the coming versions. We can divide this technique into three parts: the software metrics, the classifier and the evaluation of the classifier [10].

A software defect alludes to a defect in a system. An error is irregularity between the observed performance of a system and its specified performance. A software failure happens when the delivered product deviates from correct service and perform sudden behavior from user requirements. A software defect or error may not necessarily cause a software failure. Defect prediction is recognizing that a problem has occurred, regardless of the possibility that you don't have a clue about the reason. Defects might be predicted by a variety of quantitative or qualitative approaches. This includes a number of the multivariable, model-based approaches. Defect diagnosis is investigating at least one root causes of problems to the point where corrective action can be taken. This is additionally referred to as "defect isolation", particularly when need to demonstrate the distinction from defect prediction. A "defect" or "problem does not need to be the result of a complete failure of a software product. In a procedure plant, root causes of non-optimal operation may be hardware failures however problems may likewise be caused by poor decision of operating targets, poor feedstock quality or human error.

Software Based Defect Prediction Techniques

1. Algorithm Based Defect Tolerance (ABFT):- ABFT is used for detecting, locating, and correcting defects with a software procedure. It exploits the structure of numerical operations. This approach is effective however lacks of generality. It is appropriate for applications utilizing regular

structures, and in this manner it is used for a limited arrangement of problems.
2. Assertions:- Assertions or the logic statements embedded at various points in the program reflect invariant relationships between the variables of the program and they regularly prompt to different problems as assertions are not transparent to a programmer and their effectiveness depends on the way of an application and on a programmer's capacity.
3. Control Flow Checking (CFC):- The fundamental task of CFC is to partition an application program in essential blocks or the without branch parts of code. A deterministic signature (or number) is assigned to every block and defects are predicted by comparing the run-time signature with a pre-computed one. In most CFC strategies one of the significant problems is to tune the test granularity that ought to be used.
4. Procedure Duplication (PD):- The programmer decides to duplicate the most critical procedures and to compare the got results on executing the procedures on two distinct processors. This approach requires a programmer to choose which procedures to be duplicated and to introduce legitimate checking on the results. These code modifications are done manually and might introduce errors.
5. Error Prediction by Duplicated Instructions (EDDI):- Computation results from master and shadow instructions are compared before writing to memory. Upon mismatch, the program jumps to an error handler that will cause the program to restart. EDDI has high error coverage at the cost of performance penalty because of time redundancy as introduced into the system. Since we use general purpose registers as shadow registers, more register spilling happens with EDDI. Additional spilling causes more performance overhead since it increases the number of memory operations.
6. Software Implemented Error Prediction and Correction (EDAC):- Software Implemented EDAC approaches (e.g., Cyclic Redundancy Checks or CRC, Hamming Codes, Bose-Chaudhuri-Hocquenghem or BCH and so forth,) are effective in error prediction yet they suffer from high time overhead. Hamming, BCH and RS codes have pleasant mathematical structures. In any case, there is a limitation with regards to code lengths.
7. Periodic Memory Scrubbing:- This approach depends on periodic reloading of code on fundamental memory from an immutable memory. This is effective for protecting the code segment of Operating system and application programs. Performance penalty is because of repetitive memory reading.
8. Masking Redundancy:- This approach implies running an application in the presence of defects. Couple of processors is used to run a similar program and vote to identify errors in any single processor. Errors can be masked from application software. No software rollbacks are required to fix errors.
9. Reconfiguration:- This implies removing failed modules from the system. At the point when failure happens in a module, its impacts on the rest of the segments of the system which are isolated. A substantial number of functional modules are used, which are switched automatically to replace a failing module.
10. Replication:- This ensures reliability however is expensive regarding hardware or runtime cost. The idea is to take a majority vote on a calculation replicated N times. Its software solution requires every processor to run N copies of surrounding computations and afterward vote on the result. This backs off the computation by no less than a factor of N.
11. Restore Architecture:- Transient errors or soft errors are predicted through time redundancy in the ReStore architecture. The novelty of the ReStore architecture is the use of transient error symptoms, for example, memory protection violation and incorrect control flow etc.
12. Dual Modular Redundancy (DMR) & Backward-Error Recovery (BER) & Checkpoint:- Error is predicted through differences in execution across a dual modular redundant (DMR) processor pair. DMR is a backward-error recovery (BER) technique where two processors are used to detect errors in execution.

II. LITERATURE SURVEY

Gao K. et al. [2007] proposed that how count models based upon poisson regression model and negative binomial regression model can be used for software defect predictions. It evaluates the comparative hypothesis testing, model selection and performance evaluation for the count models [3]. Zimmermann T. et al. [2007] mapped defects from the bug database of Eclipse to source code locations. The resulting data set lists the number of pre and post release defects for every package and file in the Eclipse releases 2.0, 2.1, and 3.0 [4]. Lessmann S. et al. [2008] improved software quality and testing efficiency by constructing predictive classification models from code attributes to enable a timely identification of defect-prone modules. Several classification models have been evaluated for this task [5]. Moser R. et al. [2008] identified a comparative analysis of the predictive power of two different sets of metrics for defect prediction. It choose one set of product related and one set of process related software metrics and use them for classifying Java files of the Eclipse project as defective respective defect-free [6]. D'Ambros M. et al. [2011] described the performance of the approaches using different performance indicators: classification of entities as defect-prone or not, ranking of the entities, with and without taking into account the effort to review an entity [11]. Rawat S. et al. [2012] introduced causative factors which in turn suggest the remedies to improve software quality and productivity. The paper also showcases on how the various defect prediction models are implemented resulting in reduced magnitude of defects [13]. Yang X. et al. [2012] presented the ranking approach for allocating testing resources to software modules. In this paper predicting models can be used for predict the defects. This paper only concerned with the construction of models, which include the ranking performance measure in the objective function, perform better in predicting defect-proneness rankings of multiple modules [16]. Yang X. et al. [2015] proposed a linear LTR approach. In this paper LTR approach can be compared with different count models. LTR approach provides better results than the different count models. This approach increases the performance of software. A learning-to-rank approach to construct software defect prediction models by directly optimizing the ranking performance. It shows comparison of the learning - to-rank method against other algorithms that have been used for predicting the order of software modules according to the predicted number of defects [17].

III. LEARNING TO RANK ALGORITHM

Learning to rank method refers to machine learning techniques for training the model in a ranking task. LTR approach can be used for measure the model performance. LTR is a linear model which is used for optimizing the ranking performance directly. LTR model is mostly used as compare to other models LTR approach can be also compare with the existing non-linear models. In LTR approach trained data can be used. LTR approach cans also works on different data sets. LTR is useful for many applications in information retrieval, Natural language processing and data mining. The LTR approach obtains a linear model by optimizing the ranking performance directly. The LTR approach can work with different models. Count models can be used with the LTR approach. Different data sets can be used in LTR approach for evaluating ranking of the software defects. We provide a comprehensive evaluation and comparison of the LTR approach against more algorithms for constructing SDP models for the ranking task. In previous work LTR approach can be compared with many other methods [17].

Many learning-to-rank algorithms can fit into the above framework. Keeping in mind the end goal to better comprehend them, a categorization is performed on these algorithms.

a. The pointwise approach: The input space of the pointwise approach contains the feature vector of every single document. The output space contains the relevance degree of every single document. The hypothesis space contains functions that take the feature vector of a document as the input and predict the relevance degree of the document. The loss function examines the accurate prediction of the ground truth label for every single document. In different pointwise ranking algorithms, ranking is displayed as regression, classification, and ordinal regression.

b. The pairwise approach: The input space of the pairwise approach contains a pair of documents, both represented as feature vectors. The output space contains the pairwise preference (which takes values from $\{1,-1\}$) between every pair of documents. The hypothesis space contains bi-variate functions h that take a pair of documents as the input and output the relative request between them.

c. The listwise approach: The input space of the listwise approach contains the entire group of documents connected with query q. There are two sorts of output spaces utilized as a part of the listwise approach. For some listwise ranking algorithms, the output space contains the relevance degrees of the considerable number of documents connected with a query.

IV. BOLTZMANN LEARNING

Boltzmann machines are systems of symmetrically connected units that settle on stochastic decisions about whether to be on or off. They have a simple learning algorithm that permits them to discover complex distributions behind observed data. Learning or inference in Boltzmann machines is imperative for many scientific tasks. For inference problems, the weights on connections and thresholds are settled and are utilized to represent a cost function. Inference in the Boltzmann machines is frequently utilized as a tool for some advancement problems, including troublesome combinatorial problems that have a place with NP finish or - hard issue classes, for example, the traveling salesman issue. Learning in

Boltzmann machines requires expectations of one unit as well as correlations between two units. Accordingly, the precise estimation of the correlations is essential.

It has been realized that the linear response approximation (LRA) improves the accuracy of correlations estimated by the mean field strategy. Utilizing such approximation methods inside learning systems amounts to match the empirical moments, to those acquired by the inexact inference methods. Hence, the inexact learning algorithms, with or without the LRA, are technically in light of the concept of pseudo-moment matching. Pseudo-moment matching problems, including the learning algorithm of Boltzmann machines, have likewise been addressed by a few researchers. The accuracy of probabilistic inference systems constructed by joining the BP algorithm with the LRA is investigated and concluded that the LRA can improve the estimation of correlations by including the impacts of loops in a particular system to the BP algorithm.

The global energy, E, in a Boltzmann machine is identical

$$E = -\left(\sum_{i,j} w_{i,j} s_i s_j + \sum_i \theta_i s_i\right)$$

Where, w_{ij} is the connection strength between unit j and unit i.

s_i is the state, $s_i \epsilon \{0,1\}$, of unit i.

Θ_i is the bias of unit i in the global energy function.

Often the weights are represented in matrix form with a symmetric matrix W, with zeros along the diagonal.

V. PROPOSED METHODOLOGY

The defect prediction is the technique which is applied to predict the percentage of defects in the test cases. This work is based on to detect defects from the test cases using learn-to-rank algorithm. The learn-to-rank algorithm is based on three steps. The first step is selection of population. The second step is calculation of mutation value. The last step is calculation of fitness value. The calculation of fitness value depends upon the initial population value which is selected randomly. In this work, Back Propagation technique is applied in which system learns from the experience values and derives new values. The selection of population value is not random. It depends upon the system condition which is derived using back propagation algorithm.

a. Proposed Algorithm

```
 Init population P (t)
   evaluate P (t);
   t := 0;
Network  ConstructNetworkLayers()
 InitializeWeights(Network, test cases)
For ( i=0;i=test cases; i++)
   Select Input Pattern(Input defect values)
   Forward Propagate(p)
  Backward Propagate Error(P)
  Update Weights(P )
End
Return (P)
   while not done do
              t := t + 1;

      P' := test case P (t);

recombine P' (t);

   mutate P' (t);

      evaluate P' (t);

      P := survive P,P' (t);

   end
```

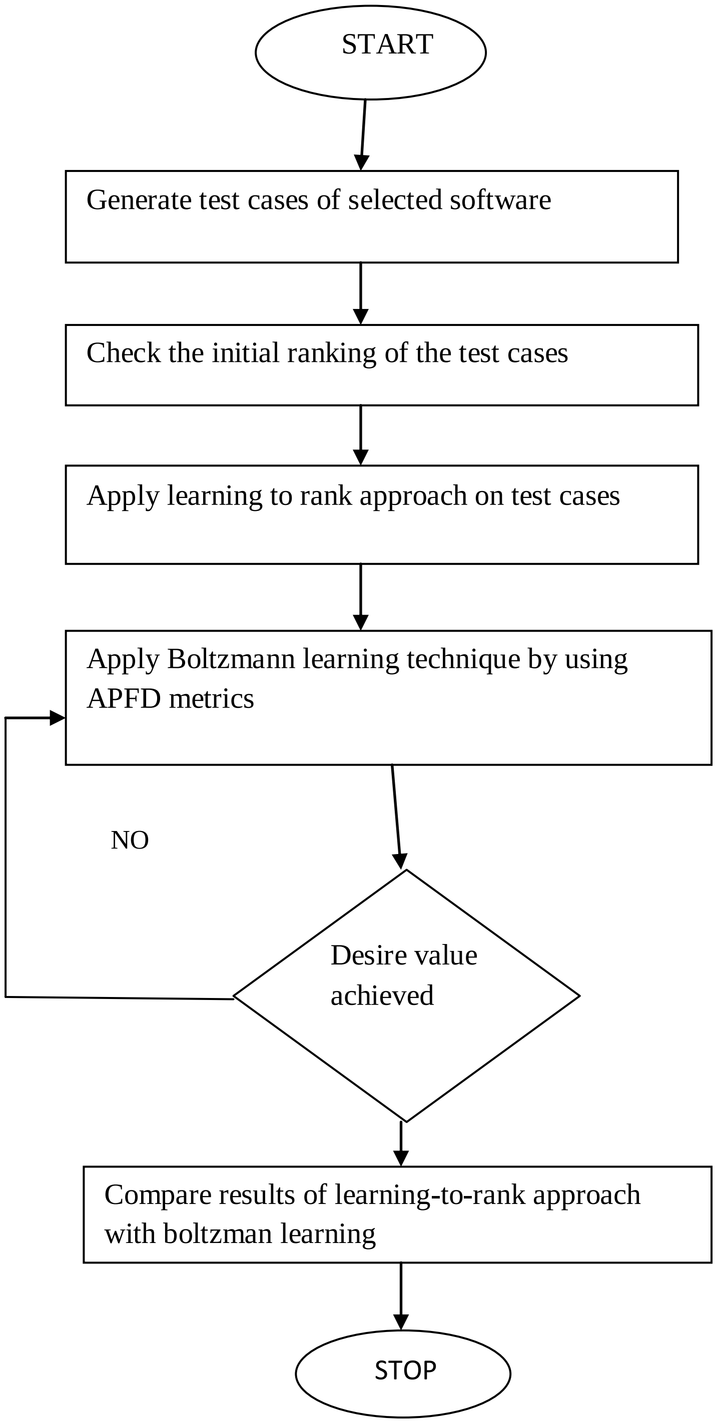
START
Generate test cases of selected software
Check the initial ranking of the test cases
Apply learning to rank approach on test cases
Apply Boltzmann learning technique by using APFD metrics
NO
Desire value achieved
Compare results of learning-to-rank approach with boltzman learning
STOP

Fig 1: Proposed Flow chart

VI. SIMULATION RESULTS

The Rank-to-rank and improved Rank-to-learn algorithms are implemented in MATLAB. The dataset is considered for the implementation which is described in the table 1

Attributes	Values
Number test cases	10
Repeated Test cases	No
Defect in the Test cases	Yes
Number of applications	1

Table 1: Properties of dataset

The proposed algorithm is implemented and interface is designed for the implementation which is described in the figures shown below

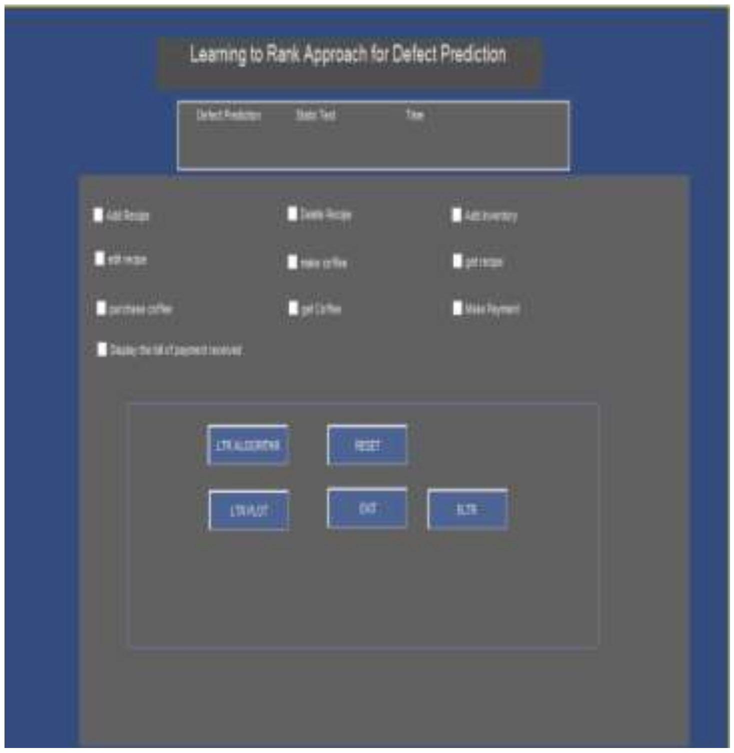

Fig 1: Interface is designed for implementation

As shown in figure 1, the interface is designed for the implementation of rank-to-learn and improved rank-learn algorithm. In the interface ten test cases are shown in which is executes existing and proposed algorithm. The result in analyzed in terms of defect prediction rate.

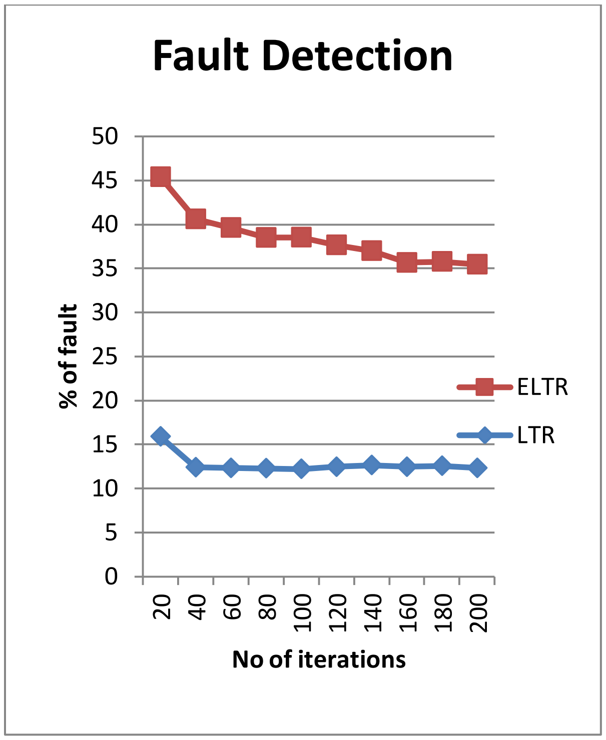

Fig 2: Comparison Graph

As illustrated in figure 2, the comparison graph is drawn between proposed and exiting algorithm. The existing algorithm is Rank-to-learn algorithm and proposed algorithm is improved Rank-to-learn algorithm. When the back propagation algorithm is implemented with Rank-to-learn algorithm the defect prediction rate is improved as shown the graph .

VII. CONCLUSION

Defect prediction is the testing technique which is applied to detect defects from the software or from the input test cases. The Rank-to-learn is the algorithm which is applied for the defects in the software. This algorithm selects population randomly which reduce defect prediction rate. In this work, technique of back propagation is applied in which system learns from the previous experiences and drive new values. This leads to improve defect prediction rate and reduce execution time . In future technique will be proposed which is based on bio-inspired techniques for the defect prediction rate

VIII. REFERENCES

[1] Graves T. L. , Karr A. F. , Marron J. S. , and Siy H. , “Predicting defect incidence using software change history,” in Proc. IEEE Trans. Softw. Eng., Vol.26, no. 7, pp. 653–661, 2000.

[2] Ostrand T. J., Weyuker E. J., and Bell R. M., “Predicting the location and number of defects in large software systems,” IEEE Trans. Softw. Eng., Vol. 31, no. 4, pp. 340–355, 2005.

[3] Gao K. and Khoshgoftaar T.M. , “A comprehensive empirical study of count models for software defect prediction,” in Proc. IEEE 28th Int. Conf. Trans. Rel., Vol. 56, no. 2, pp. 223–236, June. 2007.

[4] Zimmermann T. , Premraj R. , and Zeller A. , “Predicting defects for eclipse,” in Proc. IEEE Int. Workshop Predictor Models in Software Engineering(PROMISE'07), pp. 9–15, 2007.

[5] Jiang Y. , Cukic B. , and Ma Y. , “Techniques for evaluating defect prediction models,” in Proc. Empiric. Softw. Eng., Vol. 13, no. 5, pp. 561–595, 2008.

[6] Lessmann S. , Baesens B. , Mues C. , and Pietsch S. , “Benchmarking classification models for software defect prediction: A proposed frame work and novel findings,” in Proc. IEEE Trans. Software Engineering., Vol. 34, no. 4, pp. 485–496, 2008.

[7] Moser R. , Pedrycz W. , and Succi G. , “A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction,” in Proc. ACM/IEEE 30th Int. Conf. Software Engineering, pp. 181–190 , Dec.2008.

[8] Mende T. , and Koschke R. , “Revisiting the evaluation of defect prediction models,” in Proc. 5th Int. Conf. Predictor Models in Software Engineering, 2009, pp. 1–10.

[9] Arisholm E. , Briand L. C. , and Johannessen E. B., “A systematic and comprehensive investigation of methods to build and evaluate defect prediction models,” in Proc. J. Syst. Softw., Vol. 83, no. 1, pp. 2–17, 2010.

[10] Weyuker E.G. , Ostrand T. J. and Bell R. M. , “Comparing the effectiveness of several modeling methods for defect prediction,” in Proc. IEEE Int. J. Empiric. Softw. Eng., Vol. 15, no. 3, pp. 277–295, 2010.

[11] D'Ambros M. , Lanza M. , and. Robbes R. , “Evaluating defect prediction approaches:A benchmark and an extensive comparison,” in Proc. IEEE Conf. Softw. Eng., pp. 1–47, 2011.

[12] Wang H. , Khoshgoftaar T. M. , and Seliya N. , "How many software metrics should be selected for defect prediction," in Proc. 24th Int. Florida Artificial Intelligence Research Society Conf., pp. 69–74, 2011.

[13] Khoshgoftaar T. M. , Gao K. , and Napolitano A. , "An empirical study of feature ranking techniques for software quality prediction," in Proc. IEEE Int. J. Softw. Eng. Knowl. Eng., Vol. 22, no. 2, pp. 161–183, 2012.

[14] Wang Y. , Cai Z. , and Zhang Q. , "Differential evolution with composite trial vector generation strategies and control parameters," in Proc. IEEE Trans. Evol. Computat., Vol. 15, no. 1, pp. 55–66, 2011.

[15] Rawat S. M, Dubey K .S, "Software Defect Prediction Models for Quality Improvement: A Literature Study", in Proc. International Journal of Computer Science Issues, Vol. 9, Issue 5, No 2, September 2012 ISSN (Online): 1694-0814.

[16] Yang X. , Tang K. , and Yao X. , "A effective algorithm for constructing defect prediction models," Int. J. in Intelligent Data Engineering and Automated Learning-IDEAL, pp. 167–175, 2012.