ASSESSING THE LEVEL OF SOFTWARE RELIABILITY USING NEURO-FUZZY SYSTEM

BONTHU KOTAIAH, DR.R.A. KHAN

Abstract : This paper assess the Software Reliability in Software Systems using Neuro-Fuzzy System. The objective is to find the ways to be used to model Software Reliability i.e. to predict the severity level on the performance of the developed Software during SDLC phases. WEKA(Waikato Environment for Knowledge Analysis) is a famous collection of machine learning software written in java, developed at the University of Waikato used to assess the Software Reliability using Neuro-Fuzzy based system. The results are measured by using the metrics like Accuracy, MAE(Mean Error) and RMSE(Room Mean Squared Error).

Keywords: Reliability Prediction, Neuro-Fuzzy System, Software Fault.

Introduction : Defects in software systems continue to be a major problem. A software bug is an error, defect, mistake, or fault in a computer program that prevents it from behaving as intended (e.g., producing the unnecessary results). The developed software fault is a defect that causes software failure in an executable product. In software engineering, the deviation of software from its requirements is called a bug or defect. Majority of the bugs arise from errors and mistakes made by people in either a program's source code or during its design phase, and some are caused by compilers producing garbage code. By finding out the reasons of possible defects and by identifying software development process steps that attract more attention from the start of a project will lead to save money, time and work. This assessment of early possible faultiness of software could help on planning, controlling and executing software development activities.

When a software system is developed, the majority of faults are found in a few of its modules. In most of the cases, 55 % of faults exist within 20 % of source code. It is, therefore, much of interest is to find out fault-prone software modules at early stage of a project[14]. By using software complexity measures, which are the techniques to build models, they classify components as likely to contain faults or not. Reliability will be improved as more faults will be detected and corrected. Predicting faults early in the software life cycle can be used to improve software process control and achieve high software reliability. Timely predictions of faults in software modules can be used to direct cost-effective quality enhancement efforts to modules that are likely to have a high number of faults. Prediction models based on software metrics, can estimate number of faults in software modules.

Machine learning algorithms have been used practically in variety of application domains. The field of software engineering turns out to be a fertile ground where many software development and maintenance tasks could be formulated as learning problems and approached in terms of learning algorithms [1-4].WEKA(Waikato Environment for Knowledge Analysis) is a famous suite of machine learning techniques implementation software developed in JAVA at the University of Waikato. WEKA is available as Open Source Software under the GNU General Public License.

In this present work, the Neuro-Fuzzy Based techniques are explored and comparative analysis is performed for the prediction of faults in software systems. The paper is organized as follows: Section II explains about the methodology followed and Section III the results of the study. Finally conclusions of the research are presented in section IV.

Methodology : The first step is to find the object oriented code written by the programmers and SDLC design attributes of software systems like software metrics. The real-time defect data sets are taken from the NASA's MDP (Metric Data Program) database. The dataset collected belongs to the important secure and effective software systems being developed by NASA. The suitable metrics like product module metrics out of these data sets are considered. The term product used in this context indicating software data at individual module level. The term software metrics information applies to any finite constant values, that describe the assessed qualities of a product. The term product refers to anything to which defect data and metrics data can be associated. In most cases products will be synonymous with code related items such a functions.

The metrics are as follows:

•MODULE - This metric describes the unique numeric identifier of the product.

•LOC_BLANK - This metric describes the number of blank lines in a module i.e. the number of lines that have nothing on them at all.

•BRANCH_COUNT - This metric describes the branch count metrics i.e. the number of branches

for each module. Branches are defined as the edges that come out from a decision node. If the number of branches in program modules increases, then more testing resources like time and effort are required.

•CALL_PAIRS - This metric describes the number of calls to other functions in a module.

•LOC_CODE_AND_COMMENT- This metric specifies the number of instructions in the program which contain both code & comment in a software module.

•LOC_COMMENTS - This metric describes the number of lines of comments in a module. This includes all lines of code that are completely commented and therefore, are completely ignored by the compiler.

• CONDITION_COUNT - This metric specifies the number of test criteria included in a given software module.

•CYCLOMATIC_COMPLEXITY- This metric specifies the cyclomatic complexity of a module. It is the count of the linearly independent paths or conditions in the software. Given a control flow graph G of a program, the cyclomatic complexity V(G) can be computed as: V (G) = E - N + 2.

•CYCLOMATIC_DENSITY - This metric specifies the ratio of the individual module's cyclomatic complexity to the module length in terms of number of instructions. The desire is to extract the size component of complexity of the module. It has the effect of normalizing or reducing the complexity of a module, hence its maintenance become difficult.

•DECISION_COUNT-It specifies the count of decision points in a given software module. Decisions are performed by the conditional statements in the module.

•DECISION_DENSITY- This metric is calculated as CC/LLOC, where CC is cyclomatic complexity and LLOC is logical line of code. This metric find out the average cyclomatic density of the software code statements within the procedures or functions of your software project. Single-line procedure or function declarations are not totalled since cyclomatic complexity is not specified for them. The denominator in the ratio is the instructions or statements count code metric. A logical instruction of code is one that contains actual source code written by the programmer. An empty instruction or a comment instruction is not totaled in LLOC.

•DESIGN_COMPLEXITY - This metric specifies the design criticalness of a software module. Design criticalness is a measure of a module's SDLC decision structure as it relates to calls to other modules from different modules. This quantifies the testing effort related to integration of the individual modules.

•DESIGN_DENSITY - It describes the design density of a module and is calculated as design complexity divided by cyclomatic complexity.

•EDGE_COUNT - This metric describes the number of edges found in a given module. It represents the transfer of control from one module to another. •

•ERROR_COUNT

This metric describes the number of defects associated with a module.

•ERROR_DENSITY - This metric describes the number

of defects per 1000 lines of code for a module. It is given by,

ERROR_DENSITY=1000*(ERROR_COUNT/LOC_TOTAL).

•ESSENTIAL_COMPLEXITY - It describes the essential

complexity of a module. It quantifies the extent to which software is unstructured, providing a continuous range of object oriented quality assessments applicable to all software rather than the "all or nothing" approach of pure object oriented programming.

• ESSENTIAL_DENSITY - The Essential density is given by, (ev(G)-1)/(v(G)-1) where ev (G) stands for essential complexity and v (G) stands for cyclomatic complexity.

• LOC_EXECUTABLE - The number of lines of executable code for a module. This includes all lines of code that are not fully commented and blank. It is also known as source lines of code (SLOC). When SLOC is divided by 1000, it is called KSLOC for 1000 SLOC.

• PARAMETER_COUNT - It describes the number of parameters to a given module.

•GLOBAL_DATA_COMPLEXITY- Global Data Complexity quantifies the cyclomatic complexity of a module's structure as it relates to global/parameter data. Global data complexity measures the complexity of the global and parameter data with a module.

•GLOBAL_DATA_DENSITY - The Global Data density is calculated as: gdv(G)/v(G), i.e. dividing global data complexity by cyclomatic complexity.

• HALSTEAD_CONTENT - This metric describes the Halstead length content of a module. The Halstead measures are based on four scalar numbers derived directly from a program's source code.

n1 is the number of distinct operators,

n2 is the number of distinct operands,

N1 is the total number of operators and

N2 is the total number of operands.

Therefore, Halstead length content, n = n1 + n2.

• HALSTEAD_DIFFICULTY - The difficulty level of the module or error proneness(D) of the program modules is proportional to the number of unique

operators identified in the total program.
D = (n1 / 2) * (N2 / n2).
•HALSTEAD_EFFORT-This metric specifies the halstead effort assessment of a software module. Effort is the number of mental discriminations used by the programmer to implement the program and also the effort required to read and understand the program by the users of the program. The effort to implement (E) or understand a program is proportional to the volume and to the difficulty level of the program.
It is given by, E= D * V, where V stands for volume.
•HALSTEAD_ERROR_EST - This metric specifies the halstead defect or error estimate metric of a software module. It is an estimate or measure for the number of errors in the implemented program.
•HALSTEAD_LENGTH - This metric describes the halstead length metric of a module which includes the total number of operator occurrences and total number of operand occurrences.
• HALSTEAD_LEVEL - This metric describes the halstead level metric of a module i.e. level at which the program can be understood. The program level (L) is the inverse of the error proneness of the program. A low levelprogram is more prone to errors than a high level program. It is given by, L = 1 / D.
•HALSTEAD_PROG_TIME - This metric describes the halstead programming time metric of a module. It is estimated amount of time to implement the algorithm. The time to implement or understand a program (T) is proportional to the effort. Halstead has found that dividing the effort by 18 give an approximation for the time in seconds.It is given by, T = E / 18.
•HALSTEAD_VOLUME - This metric describes the halstead volume metric of a module that contains the minimum number of bits required for coding the program. The program volume (V) is the information contents of the program, measured in mathematical bits. It is calculated as the program length times the 2-base logarithm of the vocabulary size (n).
V = N * log2 (n)
•MAINTENANCE_SEVERITY-The Maintenance Severity is calculated as: ev (G)/v (G) i.e. essential complexity divided by cyclomatic complexity.
•MODIFIED_CONDITION_COUNT-This metric describes the number of modified conditions that exist within a module. Every condition shown to independently affect a decision outcome by varying that condition only.
•MULTIPLE_CONDITION_COUNT-This metric describes the number of multiple conditions that exist within a module.
• NODE_COUNT - This metric describes the number of nodes found in a given module. Nodes are a base metric, used to calculate many of the more involved complexity metrics.
•NORMALIZED_CYLOMATIC_COMPLEXITY -This metric describes the normalized cyclomatic complexity. Normalized complexity is simply a module's cyclomatic complexity divided by the number of lines of code in the module. This division factors the size factor out of the cyclomatic measure and identifies modules with unusually
dense decision logic. A module with dense decision logic will require more effort to maintain than the modules with less dense logic.
•NUM_OPERANDS - This metric describes the number of operands contained in a module.
•NUM_OPERATORS - This metric describes the number of operators contained in a module.
•NUM_UNIQUE_OPERANDS - This metric describes the number of unique operands contained in a module. It is a count of unique variables and constants in a module.
•NUM_UNIQUE_OPERATORS - This metric describes the number of unique operators contained in a module i.e. the number of distinct operators in a module.
• NUMBER_OF_LINES - This metric describes the number of lines in a module. Pure, simple count from open bracket to close bracket and includes every line in between, regardless of character content.
•PATHOLOGICAL_COMPLEXITY-This metric describes the pathological complexity of a module. It describes the measure of the degree to which a module contains extremely unstructured constructs. Pathological complexity measures the degree to which a module contains extremely unstructured objects.
•PERCENT_COMMENTS - This metric describes the percentage of the code that is comments. This is the number of fully commented lines of code divided by the total number of lines of code and is used as an indicator of readability.
• LOC_TOTAL - This metric describes the total number of lines for a given module. This is the sum of the executable lines and the commented lines of code. Blank lines are not counted.
In the next step the metrics are analyzed, refined and normalized. Thereafter, it is tried to evaluate the Neuro-Fuzzy Inference System for modeling of the software maintenance severity in software systems. The comparisons are made on the basis of the least value of MAE and RMSE values. Accuracy value of the prediction model is also used for comparison. The mean absolute error is chosen as the standard error. The technique having

lower value of mean absolute error is chosen as the best fault prediction technique.

Results : The performance of Adaptive Neuro Fuzzy Inference System is found to be the best out of all the hybrid NF systems [15-16] and the extra complexity in structure and computation of Mamdami based Adaptive NF Inference system with max-min composition does not necessarily imply better learning capability or approximation power [18-19]. The inference system, which is already trained, will get the metric values from the earlier stages and estimate the software maintenance severity value of the software components or modules. The Structure of Adaptive Neuro-Fuzzy Inference System is shown in fig. 1.

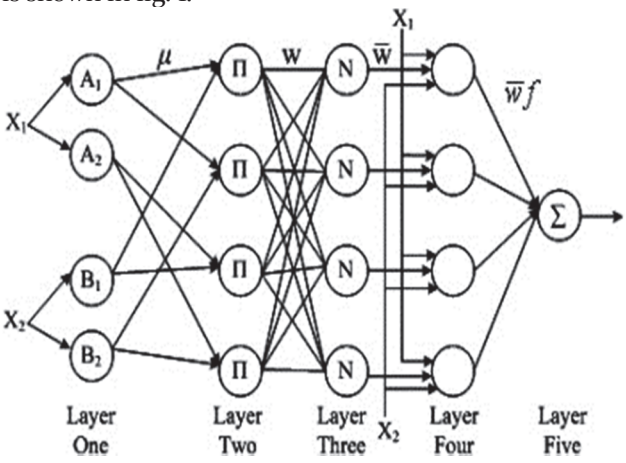

Fig.1: Structure of Adaptive Neuro-Fuzzy Inference System.

The NF system is trained using a hybrid learning algorithm using both least squares method and back propagation. In the forward pass the consequent parameters are identified using least squares and in the backward pass the premise parameters are identified using back propagation. The trained NF system is then tested for the fifteen inputs and it shows 0.1571,0.2140 and 93.3333 as MAE, RMSE and Accuracy values respectively.

Conclusion : The Neuro-fuzzy based Modeling technique has outperformed the other technique on the basis of the testing data with 0.1571, 0.2140 and 93.3333 as Mean Absolute Error, Root Mean Square Error and Accuracy values respectively. It is therefore, concluded the model is implemented and the best algorithm for classification of the software components into different level of severity of impact of the fault is found to be Neuro-Fuzzy based technique. The algorithm can be used to develop model that can be used for identifying modules that are heavily affected by the faults and those need immediate attention for debugging.

The future work can be extended in following directions:

•This work can be extended to other programming languages other than object oriented programming.

References:

1. Aha, D. and Kibler, D. (1991), "Instance-based learning algorithms",Machine Learning, vol. 6, Issue no. 1, January 1991, pp. 37-66.
2. Bellini, P. (2005), "Comparing Fault-Proneness Estimation Models",10th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS'05), vol. 0,2005,pp.205-214.
3. Challagulla, V.U.B. , Bastani, F.B. , I-Ling Yen , Paul,(2005) "Empirical assessment of machine learning based software defect prediction techniques", 10th IEEE International Workshop on Object-Oriented Real-Time Dependable Systems, WORDS 2005, 2-4 Feb 2005,pp. 263-270.
4. Evren Ceylan ,F. Onur Kutlubay, Ayse B. Bener (2006), "Software Defect Identification Using Machine Learning Techniques" Proceeding of 32nd EUROMICRO Conference on Software Engineering and Advanced Applications (EUROMICRO'06), Bogaziçi University, Turkey, pp. 240-247.
5. Fenton, N. E. and Neil, M. (1999), "A Critique of Software Defect Prediction Models", Bellini, I. Bruno, P. Nesi, D. Rogai, University of Florence, IEEE Trans. Softw. Engineering, vol. 25, Issue no. 5, pp. 675-689.

[1] Research Scholar, Babasaheb Bhimrao Ambedkar University, Lucknow, India
[2] Associative Professor, Babasaheb Bhimrao Ambedkar University, Lucknow, India
[1]kotaiah_bonthuklce@yahoo.com, [2]khanraees@yahoo.com