Vol.5, No.1 ISSN:2349-0012

INNOVATIVE RESEARCH ORGANISATION

International Journal of Advance Research in Education, Technology & Management

(Scholarly Peer Review Publishing System)

Integrated neural network and random forest based software testing automation

Sri sai college. Of engineering. And technology badhani,pathankot

Bhavna Nayyer,Anshu chopra

Abstract: The review has show that the finding the best automation software testing is still an open area of research. Software testing involves much time for its testing, thus is expensive task. The comprehensive review has shown that the role of software mining during software testing is ignored in most of existing research. The effect of software metrics during, software testing is also ignored in existing literature. Not much effort is done to predict the automation software testing, before software will be released in the market. Therefore using the machine learning techniques with the help of hybrid neural network and random forest tree based technique is proposed in this work. The use of the proposed technique is to predict the software testing effect before it release to the market, using its metrics.

Keywords: software engineering, Regression testing, Regression testing types, Test case optimization, software computing based Test case optimization

1. Introduction

In the era of software engineering, present day research community and software industry is facing big challenges due to increased maintenance cost and lack of intended software quality attributes viz. functionality, reliability, usability, efficiency, maintainability and probability. All the software quality attributes are hard to define and difficult to measure and recognize. These software quality attributes may be improved if the researchers and practitioners make use of relevant models and test process for software development. Software is a vitally important and the most essential component of any computer system. A computer cannot do any task without software. Software controls the computer and makes it do useful work. A computer without software is useless, akin to a car without someone to drive it. The range and extent of usage of software products can vary from a very simple application for day-today operations to very complex applications like aeronautics, astronautics, nuclear technology, navigation and other control operations. The companies around the world are spending a big portion of their resources to make the software systems robust, reliable, scalable, safe and secure. Software systems are not error free. Software development has long been a manual process. Therefore, human errors are possible in producing software. For example, user requirements might be misinterpreted, rules of
the system design might be violated, or there may be a careless mistake by the programmer etc. The sources of program errors are plentiful. Therefore, in spite of the best effort in the design, development and coding of software programs, some errors may remain in the software after its completion. An ever increasing fault in the software systems in the various stages of the development has kindled the interest among software enthusiasts to consider software testing more seriously. As a result, the importance of software testing as an integral part of the software development process has been steadily growing. Software testing has been regarded as the standard technique in the software industry to ensure quality and verify the correctness of the software.

2. Automatic Test Data Generation

In the direction of automate the testing process, both the generation of test data and the execution of test cases have to be automated. There are already a number of tools such as JUnit and GoboTest that automate the test case execution but the main problem lies in the automation of the test data generation. The test data generation in program testing is the process of identifying a set test data that satisfies a selected testing criterion, such as statement coverage, branch coverage, condition coverage and path coverage. Test data generation in software testing is the process of identifying program input data which satisfy selected testing criterion.A test data generator is a tool which assists a programmer in the generation of test data for a program. There are three types of test data generators: path wise test data generators, data specification generators and random test data generators.

INNOVATIVE RESEARCH ORGANISATION
International Journal of Advance Research in Education, Technology & Management
(Scholarly Peer Review Publishing System)

- Path wise test data generators**:-**Path wise test data generators are systems that test software using a testing criterion which can be path coverage, statement coverage, branch coverage, etc. The system automatically generates test data to the chosen requirements. A path wise test generator consists of a program control flow graph construction, path selection and test data generation
- Data specification generators:-Deriving test data from specification belongs to the "black-box" testing method. Such a strategy generates test cases and test data. The test data can then be applied to software and the effectiveness can be measured.
- Random test data generators:-Select random inputs for the test data from some distribution. The automatic production of random test data, drawn from a uniform distribution, should be the default method by which other systems should be judged. We focus on path wise test data generator system that test software using white-box (Structural) testing criteria as it is more widely applied. The system automatically generates test data to the chosen requirements. The first step in applying a white box testing specifically path wise test data generator is to select a test adequacy criterion, e.g. statement or branch coverage. The next step, then, is to find a set of test data that satisfies the selected adequacy criterion, which is called adequate test data. Intesting a program, adequate test data generation is the process of identifying a set of test data, which satisfies given testing criterion. Generating adequate test data manually is labors intensive and time-consuming process. This problem has motivated researchers to create test data generators that can examine a program's structure and generate adequate test data automatically. It is not a trivial task to judge whether a finite set of input test data is adequate or not. The goal is to uncover as many faults as possible with a potent set of a constrained number of tests. Obviously, a test series that has the potential to uncover many faults is better than one that can only uncover few. In general, the process of automatic structural test data generation, for path coverage, consists of three major steps:

1. Construction of control logic graph, e.g. control flow graph (CFG) or control dependence graph (CDG);
2. Path selection; and
3. Test data generation that involves dynamic execution of the target program.

The target program must be instrumented in order to monitor the assessment of testing objective when the program is executed with given input data. In most test data generator, the instrumentation is considered to be pre-process stage before the generator can actually be used. This instrumentation process is the process of inserting probes (tags) at the beginning of every block of code of interest, i.e. at the beginning/ending of each function and after the true and false outcomes of each condition. For example in path coverage, these tags are used to monitor and provide the test data generator with a feedback on the traversed path within the program while it is executed with trial test data.

3. REGRESSION TESTING

Regression testing is defined as "the process of retesting the modified parts of the software and ensuring that no new errors have been introduced into previously tested code". Regression testing is performed on modified software to ensure the correct behaviour of the software and the absence of any adverse effect of the modifications on the software quality. Regression testing is regarded as an important technique to manage the changes in the software programs. Regression testing basically executes the test suite when the original software program is modified whenever a new functionality is added or the original system is improved. In a way, Regression testing can be considered as an evaluation system for the new program which attempts to revalidate the old functionality inherited from the old version. Regression testing is an expensive but essential activity in software maintenance. Regression testing nearly consumes half of the time spent on
software maintenance activities. However improvements in the regression testing techniques have shown to improve the elapsed time as well as expenses in software maintenance.In particular, two

Vol.5, No.1 ISSN:2349-0012

INNOVATIVE RESEARCH ORGANISATION

International Journal of Advance Research in Education, Technology & Management

(Scholarly Peer Review Publishing System)

regression testing methodologies that reuse tests are considered: regression test selection and test case prioritization.

4. TYPES OF REGRESSION TESTING

Regression testing techniques can be grouped into four categories as Retest all, Regression Test Selection, Test Case Prioritization, and Hybrid Approach.

- **Retest All:-** It is one of the conventional methods of regression testing. In this method, all the test cases in the existing test suite are re-executed. Hence, this technique is very expensive as compared to techniques which will be discussed further as regression test suites are costly to execute in full as it require more time and budget.
- **Regression Test Selection (RTS):-**In the regression test selection technique, a part of test suite is rerun instead of rerunning the whole test suite if the cost of selecting a part of test suite is less than the cost of running the tests that RTS allows us to omit. Further, the existing test suite can be divided into (1) Reusable test cases; (2) Re-testable test cases; (3) Obsolete test cases. In addition to this classification RTS may create new test cases that test the program for areas which are not covered by the existing test cases.RTS techniques can be broadly classified into three categories.

➢ **Coverage techniques:-**This technique takes the test coverage criteria into account. Test cases are selected to work on the coverable parts of the programs that have been modified.

➢ **Minimization techniques:-**This technique select minimum set of test cases to perform the regression testing.

➢ **Safe techniques:-**Here the primary focus is on those test cases that produce different output with a modified program as compared to its original version. Rothermel (1996) identified the various categories in which regression test selection technique can be evaluated and compared. The categories identified were inclusiveness, precision, efficiency and generality.

- **Test Case Prioritization:-**Test case prioritization is the technique of ordering the test cases in order to increase the rate of fault detection in a test suite. There are two types of test case prioritization namely general prioritization and version specific prioritization. In case of general prioritization, the order of the selection of test cases can be effective on average subsequent versions of software. On the other hand, version specific prioritization is concerned with particular version of the software.
- **Hybrid Approach:-**The fourth type of regression technique is a hybrid approach which combines both regression test selection and test case prioritization. A large number of researchers have proposed many algorithms for hybrid approach. For example, Aggrawal et al (2004) proposed a model that achieves 100% code coverage optimally during version specific regression testing. They also proposed a test selection algorithm to support the model. Wong et al (1997) proposed a new hybrid technique that combines modification, minimization and prioritization based selection using test history. Singh et al (2006) have proposed a hybrid technique based on regression test selection and test case prioritization.

5. TEST CASE OPTIMIZATION

Test case optimization is concerned with finding the best subset of test cases from a pool of test cases to perform regression testing on the modified software program. Some of the activities included in test case optimization are classification, selection, minimization, prioritization and filtration. A brief overview of each of the optimization technique is presented here. Test cases minimization is a selection of the smallest subset the test cases from a pool of test cases to be audited for a program. Test suite reduction seeks to reduce the number of test cases in a test suite while retaining a high percentage of the original suite's fault detection effectiveness. Test suite

minimization techniques seek to reduce the effort required for regression testing by selecting an appropriate subset of test suite. Test cases selection also finds minimal cardinality subset of test cases from the pool of test cases. One major difference between test cases minimization and test case selection is that test case se-lection chooses a temporary subset of test cases, whereas test suite minimization reduces the test suite permanently based on some external criterion such as structural coverage. Test case prioritization techniques try to find an ordering/ ranking of test cases, so that some test case adequacy can be maximized as early as possible. Test case prioritization and filtration depend on the quality of initial population of test cases. Test case filtration and prioritization are closely related. In fact, test cases can be filtered by selecting the first N ordered test cases. The goal of test case filtration is to chunk/ filter out irrelevant, redundant and less fit test case from the test suite, so that a large portion of the defects would be found as if the whole test suite was to be used. It is often desirable to filter a pool of test cases for a program in order to identify a subset that will actually be executed and audited at a particular time. When it is uncertain that how many test cases can be run and audited, it is advantageous to order or rank the test cases as per priorities, so that the tester will select the test cases as per their rank or order, which permit tester to start quick, early fixing the most of the defects. Selection and prioritization of test cases are the two important solutions to the problem of test case optimization. The overall aim of the thesis is to optimize test cases for the regression testing by employing multiple factors.

6. SOFT COMPUTING TECHNIQUES BASED TEST CASE OPTIMIZATION

Fuzzy Logic

Optimization of regression test suites using fuzzy logic technique is a multi-objective optimization process and therefore it is very complex. It is considered as Black Box based testing method. Fuzzy logic has proved its worth in many other domains like communication, bio informatics, embedded applications, industrial and engineering control and network optimizations.

Case-based Reasoning

In Case Based Reasoning, problems are solved by adapting the solutions of similar previous cases stored in a case memory. The Case Based Reasoning as such can be tailored to meet the necessary condition for test suite reduction. Case based reasoning approach is based on the fact on how human beings think. Case base in adaptive test suite has the following features.Storage of test cases used by tester. Storage of testers experience in form of case in case base. Storage of test case results for the already tested set of test cases for a module. Deciding the mechanism for test case selection. Assigning priority of test cases. Deciding the modules to be tested. Making Defect predictions in projects.

Artificial Neural Network

For some kinds of software, developing test cases from output domain is more suitable than from input domain. A Neural Network model can be used for testing software functions. On the basis of the created function model, for given outputs, genetic algorithm is used to find the corresponding inputs, so that the automation of test cases generation from output domain is completed. But the entire process of generating test cases from output domain using ANN and GA is highly complex and hence may not be feasible for all applications.

Support Vector Machine

SVM Techniques are applied for solving variety of problems. It is a machine learning technique that uses supervised learning algorithms. In software testing, this SVM technique is used for finding infeasible GUI paths using graph based models of applications software. This method is very complex and time consuming.

Ant Colony Optimization

Regression testing is a crucial and often costly software maintenance activity. The regression test prioritization using ant colony optimization is considered as a time constraint prioritization problem. Ant Colony Optimization (ACO) is a technique based on the real life behavior of ants.

ACO has its real strength in the overwhelming behavior of ants looking for a path between their colony and a source of food. Ant colony optimization algorithm (ACO) is a probabilistic technique for solving computational problems which can be reduced to finding good paths through graphs. This ACO technique reorders test suites in time constraint environment. However, it is not widely used in the field of software testing due to the fact that is uses graphical representation of the software under test.

Genetic Algorithm

GA is an optimization technique which provides near optimal solution to any optimization problems. GA is applied to solve many problems like travelling

salesman problem, knapsack problem etc.. Genetic Algorithm is based on the idea on the natural evolution. The foundation of GA lies on the concept of the survival of fittest into a solution space. Each cycle of GA process includes initialization (encoding), selection based on fitness function, reproduction using crossover or mutation. The cycle is repeated till a solution is found that satisfies the minimum criteria or a fixed number of generations has been reached. This algorithm is considered as one of the powerful method for optimization problem and simple to implement and use. The table
3.1 lists various soft computing techniques and their advantages and
limitations.

7. COMPONENT-BASED SOFTWARE AND COMPONENT REUSE

Component-based software, software components and software reuse complement each other perfectly. Using software components to build CBS almost automatically leads to software reuse and trying to reuse software almost automatically evolves in the composition of software out of components. Software reuse can not be done without the involvement of any components. Reusing software has a much broader influence on software engineering than one might initially think. Not only does it influence the construction process, it fundamentally affects organizational structures and project structures, and itinfluences legal and economic issues of software engineering. For software reuse to become a matter of fact software life cycles have to be adapted accordingly, important new activities like domain analysis come into the scene. The reuse of legacy code poses new challenges. Maintaining it is hard enough, reusing it and in building new software systems. Requirements on software systems change constantly. Rebuilding new systems every time requirements change considerably is neither feasible nor economical. This study must be able to incorporate old components of systems, split them into usefiil artifacts, and combine them with new developments. It is the wrong approach to build gigantic monolithic systems that nobody fully understands and that are hard if not impossible to adapt to new environments and situations. Due to the ever changing and increasing requirements on software systems, researchers and practitioners have been struggling with the software crisis for decades. Software must be composed of components that can be reused and replaced. Instead of replacing a whole system every twenty years, researchers and practitioners have to continually add, remove and replace components to adapt a system to changing requirements. After twenty years everything in the system may be different, but this will have happened gradually with small changes that are manageable. Software reuse and software components will not solve all problems encounter in software engineering, but they will contribute to an important step towards more flexible software systems that are constantly evolving and adapting. Reusable, adaptable software components rather than large, monolithic applications are the key assets of successful software companies. To build CBS, researchers can reuse many things, for example, algorithms, designs, requirements specifications, procedures, modules, applications, design patterns, architectures. Components are artifacts that clearly identify in our software systems. They have an interface, encapsulate internal details and are documented separately. In this context it is required that components be easily combined with each other, especially without knowing from each other'sexistence. The primary intention in reusing components is that takes a component and integrates it into a software system e.g., take a procedure and use it for some computations, reuse an algorithm. But cannot simply take the algorithm and integrate it into a software system, it is necessary to implement it first. Thus reuse the idea that is described in some pseudo code and tells how can solve the problem. But to solve the problem ourselves using a specific programming language and dealing with the special characteristics of this language. This work focuses on component reuse because components are a field that promise a rich harvest in productivity through reuse.

A. Component-Based Reuse

The benefits of the Component-Based Reuse (CBR) and component approach can only be fully utilized if there is a sufficient supply of high quality software components and if they are being actually reused. Thus one of the key success st;,-. factors in CBSE is the reuse of software components. The establishment of the component technologies has led to the development of a number of component marketplaces *{ComponentSource).* These marketplaces offer a large number *ol^*., commercial and non-commercial components. Furthermore open source** community

is another large provider of reusable software components. There are*«" over 60000 Open Source Software (OSS) located on *(SourceForge.net)*, thelargest website for OSS. But despite of the component technologies and market,. places the level of reuse in software developing companies is mostly not satisfactory. In the following paragraphs this study will therefore present a number of obstacles to software reuse.

• Some developers have some kind of researcher attitude e.g. "do-it yourself.

• In many cases there's a lack of motivation to provide reusable, software components to other developers, since reusable components require more development effort than normal software and the developers are under high time pressure in their own projects. Lack of qualification i.e. there's not enough knowledge about CBSE among the developers and project managers.

• A lot of legal problems are not uhimately solved yet, e.g. who is responsible if a system fails due to the misinterpretation of the functionality of third party components.

• The basic concepts of the business processes in different companies have not yet been unified and maybe won't be in the near future which makes cross-organizational software reuse much more difficult. These obstacles can be overcome by using incentive systems to encourage
software reuse by providing additional education training on the topic of CBSE.

8. RANDOM FOREST

Random forests is a notion of the general technique of random decision forests[1][2] that are an ensemble learning method for classification, regression and other tasks, that operate by constructing a multitude of decision at training time and outputting the class that is the mode of the classes (classification) or mean prediction (regression) of the individual trees. Random decision forests correct for decision trees' habit of overfitting to their training set.

A. Features of Random Forests

- It is unexcelled in accuracy among current algorithms.
- It runs efficiently on large data bases.
- It can handle thousands of input variables without variable deletion.
- It gives estimates of what variables are important in the classification.
- It generates an internal unbiased estimate of the generalization error as the forest building progresses.
- It has an effective method for estimating missing data and maintains accuracy when a large proportion of the data are missing.
- It has methods for balancing error in class population unbalanced data sets.
- Generated forests can be saved for future use on other data.
- Prototypes are computed that give information about the relation between the variables and the classification.
- It computes proximities between pairs of cases that can be used in clustering, locating outliers, or (by scaling) give interesting views of the data.
- The capabilities of the above can be extended to unlabeled data, leading to unsupervised clustering, data views and outlier detection.
- It offers an experimental method for detecting variable interactions.

9. LITERATURE SURVEY

Kwankamol Nongpong et al. (2015) [1] As a software system evolves, its design getdeteriorated and the system becomes difficult to maintain. In order to improve such an internal quality, the system must be restructured without affecting its external behavior. The process involves detecting the design flaws (or code smells) and applying appropriate refactorings that could help remove such flaws. One of the design flaws in many object-oriented systems is placing members in the wrong class. This code smell is called Feature Envy and it is a sign of inappropriate coupling and cohesion.

V. Sales et al. (2013)[2] Methods implemented in incorrect classes are common bad smells in object-oriented systems, especially in the case of systems maintained and evolved for years. To tackle this design flaw, they propose a novel approach that recommends Move Method refactorings based on the set of static dependencies established by a method. More specifically, their approach compares the similarity of the dependencies established by a source method with the dependencies established by the methods in possible target classes.They evaluated their approach using systems from a compiled version of the Qualitas Corpus.

R. Shatnawi et al.(2011)[3] Software refactoring is a collection of reengineering activities that aims to improve software quality. They validate the proposed heuristics in an empirical setting on two open-source systems.It is found that the majority of refactoring heuristics do improve quality; however some heuristics do not have a positive impact on all software quality factors. They validated their findings on two open-source systems-Eclipse and Struts. For both systems,they found consistency between the heuristics and the actual refactorings.

N. Tsantalis et al.(2011)[4] propose the exploitation of past source code versions in order to rank refactoring suggestions according to the number, proximity and extent of changes related with the corresponding code smells. The underlying philosophy is that code fragments which have been subject to maintenance tasks in the past, are more likely to undergo changes in a future version and thus refactorings involving the corresponding code should have a higher priority. The approach has been integrated into an existing smell detection Eclipse plug-in, while the evaluation results focus on the forecast accuracy of the examined models.

R. Oliveto et al.(2011)[5] propose a novel approach to identify Move Method refactoring opportunities and remove the Feature Envy bad smell from source code. The proposed approach analyzes both structural and conceptual relationships between methods and uses Relational Topic Models to identify sets of methods that share several responsabilities, i.e., 'friend methods'. The analysis of method friendships of a given method can be used to pinpoint the target class (envied class) where the method should be moved in. The results of a preliminary empirical evaluation indicate that the proposed approach provides meaningful refactoring opportunities.

B. Dit et al.(2013)[6] Feature location is the activity of identifying an initial location in the source code that implements functionality in a software system. Many feature location techniques have been introduced that automate some or all of this process, and a comprehensive overview of this large body of work would be beneficial to researchers and practitioners. This paper presents a systematic literature survey of feature location techniques. Eighty-nine articles from 25 venues have been reviewed and classified within the taxonomy in order to organize and structure existing work in the field of feature location.

G. Gui et al.(2009)[7] proposes a set of new static metrics of coupling and cohesion developed to assess the reusability of Java components retrieved from the Internet by a software component search engine. These metrics differ from the majority of established metrics in three respects: they measure the degree to which entities are coupled or resemble each other, they quantitatively take account of indirect coupling and cohesion relationship and they also reflect the functional complexity of classes and methods. An empirical comparison of the new metrics with eight established metrics is described.

J. Dexun et al. (2012)[8] Weight based distance metrics and relevant conceptions are introduced in this paper, and the automatic approach for bad smells detection is proposed based on Jaccard distance. The conception of distance between entities and classes is defined and relevant computing formulas are applied in detecting. New weight based distance metrics theory is proposed to detect feature envy bad smell. This improved approach can express more detailed design quality and invoking relationship than the original distance metrics theory. With these improvements the automation of bad smells detection can be achieved with high accuracy. And then the approach is applied to detect bad smells in JFreeChart open source code. The experimental results show that the weight based distance metrics theory can detect the bad smell more accurately with low time complexity.

10. RESEARCH MTHODOLOGY

The various steps required to automatically test the software before its release.

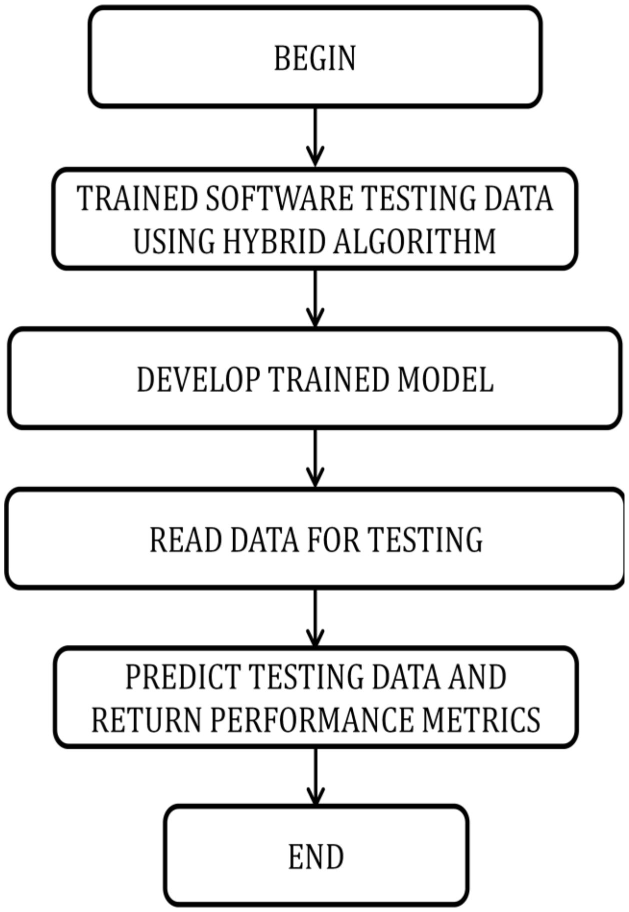
BEGIN
TRAINED SOFTWARE TESTING DATA USING HYBRID ALGORITHM
DEVELOP TRAINED MODEL
READ DATA FOR TESTING
PREDICT TESTING DATA AND RETURN PERFORMANCE METRICS
END

Fig 1 Steps of proposed work

11. Performance Evaluation

In this we compare the correlation parameter r of linear model, decision tree, random forest, neural network of existing system with proposed system which uses genetic algorithm that is it will gives best values in comparison to others.

1. PSNR (Peak Signal to Noise Ratio): Maximum Top sq sound percentage could be the percentage involving the most probable price of the signal and the ability of the corrupting noise. It is calculated in decibels (db). It could be discussed as:

$$\text{PSNR} = 10.\log_{10}\left(\frac{MAX_I^2}{MSE}\right) \quad (1)$$

2. Accuracy: Accuracy" blows here. It is not to be confused with the music. **Detail** is an explanation of arbitrary mistakes, a calculate of mathematical variability. **Accuracy** has two meanings:

1. more typically, it is an explanation of *systematic mistakes*, a calculate of mathematical error;

2. instead, the ISO describes accuracy as describing equally types of observational problem above (preferring the term *trueness* for the most popular meaning of accuracy).

$$\text{Accuracy}= \frac{\text{number of true positives+number of true negatives}}{\text{number of true positives+false positive+false negatives+true negatives}}$$
(2)

3. Error rate: Error rate of a conversation channel. The frequency with which mistakes or sound are introduced to the channel. Problem charge may be tested with regards to erroneous bits acquired per bits transmitted. For instance, one or two mistakes per 100 000 bits may be considered a common charge for a slender group point-to-point line. The circulation of mistakes is usually non standard: mistakes tend ahead in bursts. Therefore the mistake charge of a place may be given with regards to percentage of error-free seconds. one charge is stated as an adverse energy of five: one charge of just one single bit per 100 000 will be stated being an mistake charge of 10−5.

$$\text{Bit-Error-rate(BER)}:= \frac{1}{2}\ \text{efficiency}\left(\sqrt{\frac{\pounds_c}{M_\theta}}\right) \quad (3)$$

Another method of presenting error rate is to think about the errors as caused by adding the info signal to an underlying error signal.

TABLE 1: Accuracy, PSNR, Error Rate Comparison Table

ALGORITHM	ACCURACY	PSNR	ERROR RATE
DECISION TREE	49.74	34.99084	50.26
LINEAR MODEL	36.17	32.30039	63.83
NEURAL NETWORK	37.48	32.30039	63.83
RANDOM FOREST	37.34	32.40028	62.66

PROPOSED	66.44	50.61523	33.56

Table1 showing the analysis Accuracy, PSNR, Error Rate respectively as it is known that needs to be maximized. So the values of proposed approach are maximized as shown in figure 5.1.

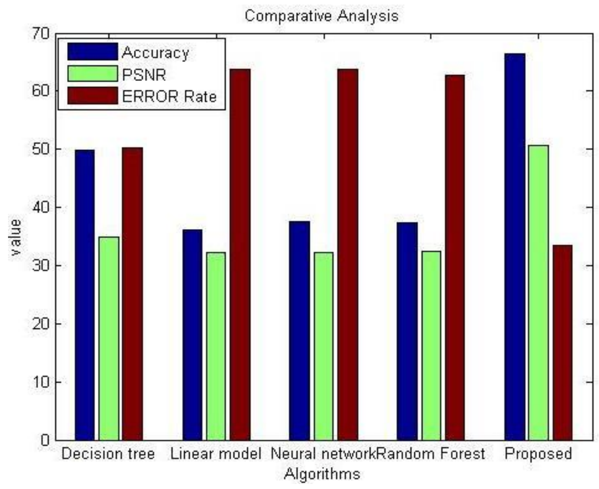

Fig 2: Accuracy, PSNR, Error Rate graph

4. Correlation coefficient(r): The correlation coefficient is just a evaluate that decides the amount to which two variables movements are associated. The range of values for the correlation coefficient is -1.0 to 1.0. If a calculated correlation is greater than 1.0 or less than -1.0, an error has been made. A correlation of -1.0 suggests a perfect negative correlation, while a correlation of 1.0 suggests an ideal good correlation.

$$N_{yx}=\frac{cov(p_y,p_x)}{\delta_y\delta_x} \quad (4)$$

5. Coefficient of determination(R): The coefficient of determination of a linear regression design could be the quotient of the variations of the repaired values and seen values of the dependent variable. When we denote z_i considering that the seen values of the dependent variable, as its suggest, and considering that the repaired cost, then the coefficient of perseverance is:

$$r^2 = \frac{\sum(\hat{z}_i-\bar{z})^2}{\sum(z_i-\bar{z})^2} \quad (5)$$

6. RMSE (Root Mean Square Error): Root-mean-square error is a measure of the differences between values predicted by a model or an estimator and the values actually observed. It can be explained as:

$$\text{RMSE} = \sqrt{\frac{1}{SN}\sum_{j=1}^{S}\sum_{i=1}^{N}(f(j,i)-f'(j,i))^2} \quad (6)$$

7. MSE(Mean Square Error): The MSE assesses the caliber of an **estimator** (i.e., a mathematical purpose mapping a style of information to a parameter of the population from that the info is sampled) or a **predictor** (i.e., a purpose mapping arbitrary inputs to a sample of prices of some arbitrary variable). Classification of an MSE varies centered on whether one is describing an estimator or even a predictor. The MSE of the predictor might be estimated by:

$$MSE = \left(\frac{1}{n}\sum_{j=1}^{n}(\widehat{Z_j} - Z_j)^2\right) \quad (7)$$

i.e., the MSE is the *mean* $(\frac{1}{n}\sum_{j=1}^{n}.)$of *the square of the problems* ($\widehat{Z_j} - Z_j^2$).).This is an simply computable amount for a specific test (and thus is sample-dependent).

TABLE 2: r, R, RMSE, MSE Comparison Table

ALGORITHM	r	R	RMSE	MSE
DECISION TREE	0.52	0.27	1.16	1.35
LINEAR MODEL	0.25	0.06	1.42	2.02
NEURAL NETWORK	0.24	0.06	1.41	2.02
RANDOM FOREST	0.26	0.07	1.41	1.99
PROPOSED	0.94	0.88	0.36	0.13

Table 2 showing the analysis r, R, RMSE, MSE respectively as it is known that needs to be minimized. So the values of proposed approach are minimized as shown in figure 3.

Vol.5, No.1 ISSN:2349-0012

INNOVATIVE RESEARCH ORGANISATION

International Journal of Advance Research in Education, Technology & Management

(Scholarly Peer Review Publishing System)

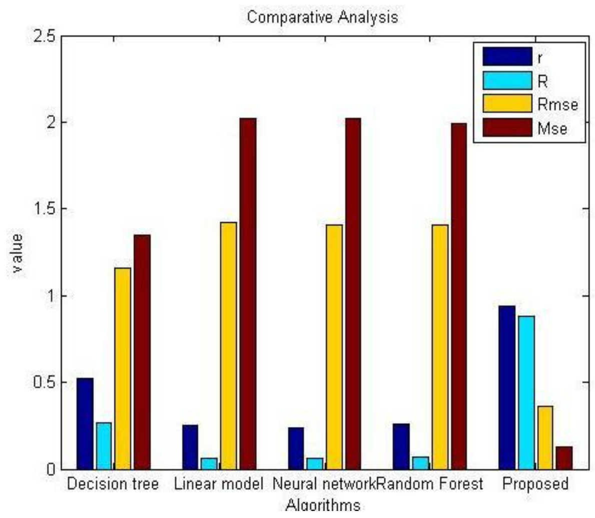

Fig 3: r, R, RMSE, MSE graph

Conclusion

In this paper, we have evaluated the performance of linear regression and Neural network based software testing. Also propose a hybrid Neural network and random forest based technique to effectively test the software before its release. Then compare existing and proposed technique using following metrics:- Accuracy(Acc), PSNR, Error Rate, Correlation(r), Coefficient of determination(R), Root mean squared error, and mean squared error. The comparison has proved that the proposed work results are much better than the existing results.

References

[1] K. Nongpong, "Feature envy factor: A metric for automatic feature envy detection," *Knowledge and Smart Technology (KST), 2015 7th International Conference on*, Chonburi, 2015, pp. 7-12.

[2] V. Sales, R. Terra, L. F. Miranda and M. T. Valente, "Recommending Move Method refactorings using dependency sets," *2013 20th Working Conference on Reverse Engineering (WCRE)*, Koblenz, 2013, pp. 232-241.

[3] Shatnawi, Raed, and Wei Li. "An empirical assessment of refactoring impact on software quality using a hierarchical quality model." International Journal of Software Engineering and Its Applications 5.4 (2011): 127-149.

[4] N. Tsantalis and A. Chatzigeorgiou, "Ranking Refactoring Suggestions Based on Historical Volatility," *Software Maintenance and Reengineering (CSMR), 2011 15th European Conference on*, Oldenburg, 2011, pp. 25-34.

[5] R. Oliveto, M. Gethers, G. Bavota, D. Poshyvanyk and A. De Lucia, "Identifying method friendships to remove the feature envy bad smell: NIER track," *2011 33rd International Conference on Software Engineering (ICSE)*, Honolulu, HI, 2011, pp. 820-823.

[6] Dit, Bogdan, et al. "Feature location in source code: a taxonomy and survey." Journal of Software: Evolution and Process 25.1 (2013): 53-95.

[7] Gui, and Paul D. Scott. "Measuring software component reusability by coupling and cohesion metrics." Journal of computers 4.9 (2009): 797-805.

[8] J. Dexun, M. Peijun, S. Xiaohong and W. Tiantian, "Detecting Bad Smells with Weight Based Distance Metrics Theory," *Instrumentation, Measurement, Computer, Communication and Control (IMCCC), 2012 Second International Conference on*, Harbin, 2012, pp. 299-304

[9] M. Goldstein and I. Segall, "Automatic and Continuous Software Architecture Validation," *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Florence, 2015, pp. 59-68.

[10] S. Fu and B. Shen, "Code Bad Smell Detection through Evolutionary Data Mining," *2015 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, Beijing, 2015, pp. 1-9.

[11] A. Yamashita, "How Good Are Code Smells for Evaluating Software Maintainability? Results from a Comparative Case Study," *Software Maintenance (ICSM), 2013 29th IEEE International Conference on*, Eindhoven, 2013, pp. 566-571.

[12] Meyers, Timothy M., and David Binkley. "Slice-based cohesion metrics and software intervention." Reverse Engineering, 2004.

Proceedings. 11th Working Conference on. IEEE, 2004.

[13] N. Tsantalis, T. Chaikalis and A. Chatzigeorgiou, "JDeodorant: Identification and Removal of Type-Checking Bad Smells," *Software Maintenance and Reengineering, 2008. CSMR 2008. 12th European Conference on*, Athens, 2008, pp. 329-331.