STM JOURNALS

Journal of Artificial Intelligence Research & Advances
ISSN: 2395-6720
Volume 7, Issue 1
www.stmjournals.com

Parameter Estimation of Software Reliability Growth Model

Bisma Gulzar Mir[1,*], Sheikh Riyaz-ul-Haq[2]
[1]Student, Department of Information Technology, Central University of Kashmir, Ganderbal, Jammu and Kashmir, India
[2]Assisant Professor, Department of Information Technology, Central University of Kashmir, Ganderbal, Jammu and Kashmir, India

Abstract

Reliability normally covers each and every part of the device thinking about hardware, software program and processes but software program reliability paperwork an important factor as we are continually worried with efficient and quality software program. Software reliability is a crucial characteristic of software, nice for predicting the diploma of creditability of the precise software for the specified time frame and in particular surroundings. Software reliability models are categorized into two; one is the static version in which modeling and evaluation of software logic is performed at the same code, different one is dynamic model that observes the brief behavior of debugging technique in the course of checking out section. But the model parameters are in nonlinear relationships which create many troubles in locating parameters which might be best using traditional techniques like maximum likelihood and least rectangular estimation. Various algorithms have been introduced which makes the task of parameter estimation less complicated. Parameter estimation of Non-homogenous Poisson Procedure (NHPP) fashions, the use of MLE (Maximum likelihood Estimation) and seek algorithms referred to as particle swarm optimization has been explored in this paper. The incorporation of different device learning procedures within the prediction of software program reliability has proven enhancements in comparison to the statistical strategies.

Keywords: *Software reliability, Support vector machine Software reliability growth models, Maximum likelihood Estimation, Particle Swarm Intelligence, parameter estimation*

****Author for Correspondence*** E-mail: bismagulzzar.bg@gmail.com

INTRODUCTION

Software permeates our daily life. There is most likely no other human-made material which is more inescapable than programming in our advanced society. It has turned into a pivotal piece of numerous parts of society: home apparatuses, media communications, autos, planes, shopping, evaluating, web instructing, individual entertainment, etc. Specifically, science and technology request astounding programming for making upgrades and achievements. The size and complexity of software systems have developed drastically during a previous couple of decades, and the pattern will unquestionably proceed later on. The information from industry demonstrates that the size of the product for different frameworks and applications has been developing exponentially for as long as 40 years [1]. The pattern of such development in the media transmission, business, barrier, and transportation enterprises demonstrates a compound development rate of multiple times like clockwork. As a result of this consistently expanding reliance, programming disappointments can prompt genuine, even deadly, outcomes in security basic frameworks just as in typical business. Past programming disappointments have hindered a few high perceivability programs and have prompted loss of business [2]. The ubiquitous software is also invisible, and its invisible nature makes it both beneficial and harmful. From the positive side, systems around us work seamlessly thanks to the smooth and swift execution of software. From the negative side, often do not

know when, where and how software ever has failed, or will fail. Consequently, while reliability engineering for hardware and physical systems continuously improves, reliability engineering for software does not really live up to our expectation over the years. The unsettled software crisis presents huge open doors for programming designing analysts just as specialists. The capacity to oversee quality programming generation is not just a need, yet in addition a key distinctive factor in keeping up an upper hand for current organizations. Programming unwavering quality designing is fixated on a key characteristic, programming dependability, which is characterized as the likelihood of disappointment free programming task for a predetermined timeframe in a predefined domain [3]. Among different characteristics of programming quality, for example, usefulness, ease of use, capacity, and practicality, and so forth, programming dependability is commonly acknowledged as the main consideration in programming quality since it measures programming disappointments, which can make a ground-breaking framework broken. Software reliability engineering (SRE) is consequently characterized as the quantitative investigation of the operational conduct of programming-based frameworks regarding client necessities concerning unwavering quality. As a demonstrated method, SRE has been embraced either as standard or as a best current practice by in excess of 50 associations in their product undertakings and reports [4], including AT&T, Lucent, IBM, NASA, Microsoft, and numerous others in Europe, Asia, and North America. Be that as it may, this number is still generally little contrasted with the enormous measure of programming makers on the planet.

Existing SRE strategies experience the ill effects of various weaknesses. Most importantly, current SRE procedures gather the disappointment information during incorporation testing or framework testing stages. Failure data gathered during the late testing stage might be too late for fundamental design changes. Besides, the disappointment information gathered in the in-house testing might be constrained, and they may not speak to disappointments that would be revealed under real operational condition. This is particularly valid for astounding programming frameworks which require broad and wide-running testing. The unwavering quality estimation and expectation utilizing the confined testing information may cause precision issues. Thirdly, current SRE procedures or demonstrating strategies depend on some unreasonable suspicions that take the dependability estimation excessively hopeful in respect to genuine circumstances. Obviously, the current programming unwavering quality models have had their victories; yet every model can discover fruitful cases to legitimize its reality.

SOFTWARE FAILURE MECHANISMS

Software failures might be because of blunders, ambiguities, oversights or confusion of the detail that the product should fulfill, thoughtlessness or ineptitude recorded as a hard copy code, insufficient testing, wrong or sudden use of the product or other unanticipated issues. Equipment deficiencies are for the most part physical flaws, while programming issues are configuration shortcomings, which are more earnestly to imagine, arrange, distinguish, and right [3]. Configuration issues are firmly identified with fluffy human components and the plan procedure, which we do not have a strong comprehension. An incomplete rundown of the unmistakable attributes of programming contrasted with equipment is recorded beneath:

a) Failure cause: Software defects are mainly design defects.

b) Wear-out: Software does not have energy related wear-out phase. Errors can occur without warning.

c) Repairable system concept: Periodic restarts can help fix software problems.

d) Time dependency and life cycle: Software reliability is not a function of operational time.

e) Environmental factors: Do not affect software reliability, except it might affect program inputs.

f) Reliability prediction: Software reliability cannot be predicted from any physical basis, since it depends completely on human factors in design.

g) Redundancy: Cannot improve software reliability if identical software components are used.

h) Interfaces: Software interfaces are purely conceptual other than visual.

i) Failure rate motivators: Usually not predictable from analyses of separate statements.

j) Built with standard components: Well-understood and extensively tested standard parts will help improve practicality and unwavering quality. Be that as it may, in programming industry, we have not watched this pattern. Code reuse has been around for quite a while, yet to a constrained degree. Carefully talking there are no standard parts for programming, aside from some institutionalized rationale structures.

Accomplishing profoundly solid programming from the client's point of view is a requesting work for all product specialists and unwavering quality designers. [2] Summarizes the accompanying four specialized zones which are pertinent to accomplishing dependable programming frameworks and they can likewise be viewed as four issue lifecycle systems:

1) Fault prevention: to avoid, by construction, fault occurrences.
2) Fault removal: to detect, by verification and validation, the existence of faults and eliminate them.
3) Fault tolerance: to provide, by redundancy, service complying with the specification despite faults having occurred or occurring.
4) Fault/failure forecasting: to estimate, by evaluation, the presence of faults and the occurrences and consequences of failures. This has been the main focus of software reliability modeling.

TIME INDEX

In hardware, materials break down after some time. Henceforth, schedule time is a broadly acknowledged record for unwavering quality capacity. In programming, disappointments will never occur if the program is not utilized. With regards to programming dependability, time is more suitably deciphered as the pressure set on, or measure of work performed by, the product. The accompanying time units are commonly acknowledged as lists of the product unwavering quality capacity:

- **Execution time:** CPU time; time during which the CPU is busy.
- **Operation time:** Time the software is in use.
- **Calendar time:** Index used for software running 24 hours a day.
- **Run:** A job submitted to the CPU.
- **Instruction:** Number of instructions executed.
- **Path:** The execution sequence of an input.

SOFTWARE RELIABILITY MODELS AND MEASUREMENT

A software reliability model indicates the type of an arbitrary procedure that depicts the conduct of programming disappointments regarding time. A verifiable survey just as an application viewpoint of programming unwavering quality models can be found by Cheng et al. [5]. There are three primary unwavering quality displaying approaches: the mistake seeding and labeling approach, the information area approach, and the time-space approach, which is viewed as the most well-known one. The basic principle of time domain software reliability modeling is to perform curve fitting of observed time-based failure data by a pre-specified model formula, such that the model can be parameterized with statistical techniques (such as the Least Square or Maximum Likelihood methods). The model can then provide estimation of existing reliability or prediction of future reliability by extrapolation techniques. Software reliability models usually make several common assumptions, as follows: (1) The activity condition where the dependability is to be estimated is equivalent to the testing condition in which the unwavering quality model has been parameterized. (2) Once a disappointment happens, the shortcoming which causes the disappointment is quickly evacuated. (3) The

flaw expulsion procedure would not present new blames. (4) The quantity of issues natural in the product and the way these deficiencies show themselves to cause disappointments pursue, at any rate in a factual sense, certain numerical formulae. Since the quantity of shortcomings (just as the disappointment rate) of the product framework decreases when the testing advances, bringing about development of dependability, these models are regularly called programming unwavering quality development models (SRGMs).

The advantages of such modeling approaches are: (1) The physical meaning of the failure mechanism can be properly interpreted, so that the effect of failures on reliability, as measured in the form of failure rates, can be directly applied to the reliability models. (2) The combination of hardware reliability and software reliability to form system reliability models and measures can be provided in a unified theory. Even though the actual mechanisms of the various causes of hardware faults and software faults may be different, a single formulation can be employed from the reliability modeling and statistical estimation viewpoints. (3) System reliability models inherently engage system structure and modular design in block diagrams. The resulting reliability modeling process is not only intuitive (how components contribute to the overall reliability can be visualized), but also informative (reliability-critical) components can be quickly identified. Among all software reliability models, SRGM is probably one of the most successful techniques in the literature, with more than 100 models existing in one form or another, through hundreds of publications. In practice however, SRGMs encounter major challenges. First of all, software testers seldom follow the operational profile to test the software, so what is observed during software testing may not be directly extensible for operational use. Secondly, when the number of failures collected in a project is limited, it is hard to make statistically meaningful reliability predictions. Thirdly, some of the assumptions of SRGM are not realistic, e.g., the assumptions that the faults are independent of each other; that each fault has the same chance to be detected in one class; and that correction of a fault never introduces new faults [6]. Nevertheless, the above setbacks can be overcome with suitable means. Given proper data collection processes to avoid drastic invalidation of the model assumptions, it is generally possible to obtain accurate estimates of reliability and to know that these estimates are accurate. Although some historical SRGMs have been widely adopted to predict software reliability, researchers believe they can further improve the prediction accuracy of these models by adding other important factors which affect the final software quality [7]. Among others, code coverage is a metric commonly engaged by software testers, as it indicates how completely a test set executes a software system under test therefore, influencing the resulting reliability measure. To incorporate the effect of code coverage on reliability in the traditional software reliability models, Chen et al. [7] proposes a technique using both time and code coverage measurement for reliability prediction. It reduces the execution time by a parameterized factor when the test case neither increases code coverage nor causes a failure. These models, known as adjusted Non-Homogeneous Poisson Process (NHPP) models, have been shown empirically to achieve more accurate predictions than the original ones.

Metrics and Measurements

Metrics and measurements have been an important part of the software development process, not only for software project budget planning but also for software quality assurance purposes. As software complexity and software quality are highly related to software reliability, the measurements of software complexity and quality attributes have been explored for early prediction of software reliability [8]. Static as well as dynamic program complexity measurements have been collected, such as lines of code, number of operators, relative program complexity, functional complexity, operational complexity, and so on. The complexity metrics can be further included in software reliability models for early reliability prediction, for

example, to predict the initial software fault density and failure rate. In SRGM, the two measurements related to reliability are: 1) the number of failures in a time period; and 2) time between failures. An important advancement of SRGM is the notation of "time" during which failure data are recorded. It is demonstrated that CPU time is more suitable and more accurate than calendar time for recording failures, in which the actual execution time of software can be faithfully represented [4]. More recently, other forms of metrics for testing efforts have been incorporated into software reliability modeling to improve the prediction accuracy [9]. One key problem about software metrics and measurements is that they are not consistently defined and interpreted, again due to the lack of physical attributes of software. The achieved reliability measures may differ for different applications, yielding inconclusive results. A unified ontology to identify, describe, incorporate and understand reliability-related software metrics is therefore, urgently needed.

Data Collection and Analysis

The software engineering process is described sardonically as a garbage-in/garbage-out process. That is to say, the accuracy of its output is bounded by the precision of its input. Data collection, consequently, plays a crucial role for the success of software reliability measurement. Given field failure data collected from a real system, the analysis consists of five steps: 1) preprocessing of data, 2) analysis of data, 3) model structure identification and parameter estimation, 4) model solution, if necessary, and 5) analysis of models.

In Step 1, the necessary information is extracted from the field data. The processing in this step requires detailed understanding of the target software and operational conditions.

In Step 2, the data are interpreted. Typically, this step begins with a list of measures to evaluate. However, new issues that have a major impact on software reliability can also be identified during this step. The results from Step 2 are reliability characteristics of operational software in actual environments and issues that must be addressed to improve software reliability. These include fault and error classification, error propagation, error and failure distribution, software failure dependency, hardware-related software errors, evaluation of software fault tolerance, error recurrence, and diagnosis of recurrences.

In Step 3, appropriate models (such as Markov models) are identified based on the findings from Step 2. We identify model structures and realistic ranges of parameters. The identified models are abstractions of the software reliability behavior in real environments. Statistical analysis packages and measurement-based reliability analysis tools are useful at this stage.

Step 4 involves either using known techniques or developing new ones to solve the model. Model solution allows us to obtain measures, such as reliability, availability, and perform ability. The results obtained from the model must be validated against real data.

In Step 5, "what if" questions are addressed, using the identified models. Model factors are varied and the resulting effects on software reliability are evaluated. Reliability bottlenecks are determined and the effects of design changes on software reliability are predicted. Research work currently addressed in this area includes software reliability modeling in the operational phase, the modeling of the impact of software failures on performance, detailed error and recovery processes, and software error bursts. The knowledge and experience gained through such analysis can be used to plan additional studies and to develop the measurement techniques.

PARAMETER ESTIMATION TECHNIQUES

The parameter estimation strategies normally entails becoming the sample facts into underlying version which affords the expected values for the unknown parameters gift inside the statistical model used. Barnard and Bayes [10] depicted that for maximum mathematical model, if the no. of parameters is massive and is located facts is erroneous, calibrations can be accomplished in most likelihood (ML)

framework, where the nation of estimate is the parameter which maximizes the chance function. Knafl [11] proposed the life situations for optimum probability parameter estimates for numerous typically hired two-parameter software program reliability fashions. For those fashions, the most probability equations were expressed in terms single equation in a single unknown. There is a no. of strategies for fitting the pattern statistics for those models. The most not unusual approach for the direct parameter estimation is most likelihood technique (MLE). It's the technique that is used to determine the parameters of statistical fashions. The second method is fitting the curve described by way of the characteristic to the facts and estimating the parameters from the quality suit to the curve. The most common technique for the oblique parameter estimation is the least rectangular approach (LSE). Although methods like MLE and LSE provides the way to estimate the parameters; however, the parameters are inside the nonlinear relationship this makes them to suffer from no. of issues finding the most excellent parameters to tune the model for better estimation. For optimized estimation of parameters of nonlinear systems, diverse seek algorithms have been used and have provided fine results.

EXPERIMENTAL EVALUATION

Parameters of Goel Okumoto model has been estimated for two datasets. Figure 1 represents the results of the dataset [12] which contains real measured data of a real time control application. Figure 1 shows curve in which the root mean square error is minimized at each iteration.

Figure 2 shows the curve for dataset 2 in which error is reduced at each iteration.

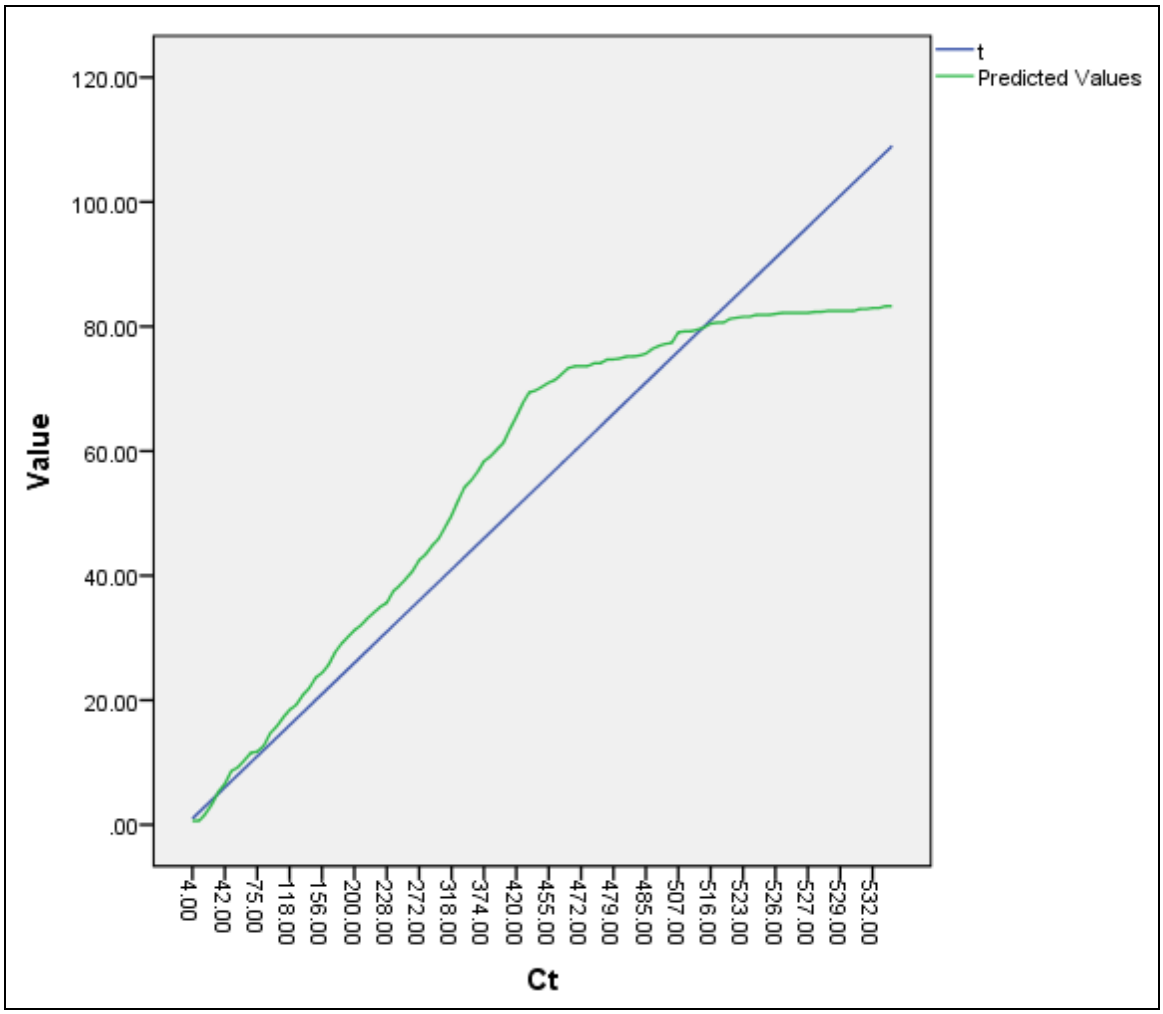

Fig. 1: *Actual and Predicted Faults for Dataset 1.*

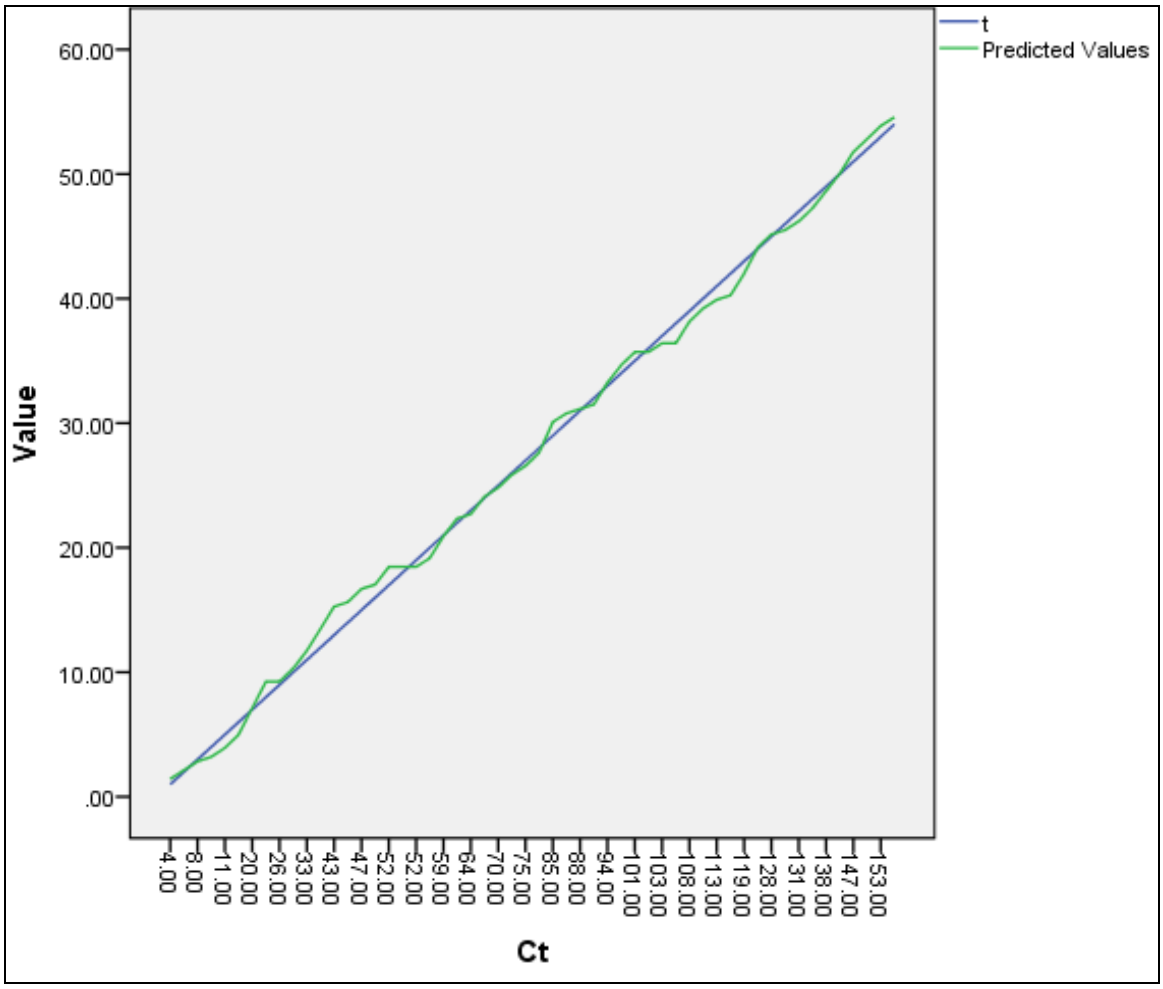

Fig. 2: *Actual and Predicted Faults for Dataset 2.*

CONCLUSION

In this paper, we mentioned numerous methods of predicting the software reliability. When the software gadget is designed, the essential challenge is the quality of the software program device. The satisfactory of software gadget relies upon on various factors like fee reliability, efficiency and so on. In this paper, we explored particle swarm optimization techniques to estimate the parameters of software program reliability models. It has been explored that fixing optimization issues the usage of evolutionary algorithms like particle swarm optimization is discovered to be extra reliable and computationally simpler as evolutionary techniques like swarm intelligence have shown better outcomes than that of traditional methods. As the cost of software application failures grows and as these failures increasingly impact business performance, software reliability will become progressively more important. Employing effective software reliability engineering techniques to improve product and process reliability would be the industry's best interests as well as major challenges. Traditional techniques used to assess the reliability of these models like MLE and LSE suffered from a number of limitations. We have shown various approaches to access the reliability of software system. The entire system of software reliability is considered useful for software development and testing industry.

ACKNOWLEDGEMENT

I would like to pay my sincere gratitude to Department of Information technology Central University of Kashmir, especially Mr. Afaq Alam Khan (Assistant Professor), Mr Zahoor Ahmad Najar (Assistant Professor), for their valuable time they used to dedicate and all the

time enlightening me about the new horizons of research related to computer engineering and its allied areas.

REFERENCES

1. Humphrey WS. *The Future of Software Engineering: I*, Watts New Column, News at SEI, 4(1), March, 2001.
2. Lyu MR. (ed.), *Handbook of Software Reliability Engineering*, IEEE Computer Society Press and McGraw- Hill, 1996.
3. ANSI/IEEE, Standard Glossary of Software Engineering Terminology, *STD-729-1991, ANSI/IEEE*, 1991.
4. Musa JD. *Software Reliability Engineering: More Reliable Software Faster and Cheaper (2nd Edition)*, Author House, 2004.
5. Cheng J, Bell DA, Liu W. Learning Belief Networks from Data: An Information Theory Based Approach, *Proceedings of the Sixth International Conference on Information and Knowledge Management*, Las Vegas, 1997, 325–331p.
6. Shooman ML. *Reliability of Computer Systems and Networks: Fault Tolerance, Analysis and Design*, Wiley, New York, 2002.
7. Chen M, Lyu MR, Wong E. Effect of Code Coverage on Software Reliability Measurement, *IEEE T Reliab*. 50(2), June 2001, 165–170p.
8. Rome Laboratory (RL), *Methodology for Software Reliability Prediction and Assessment*, Technical Report RLTR- 92-52, volumes 1 and 2, 1992.
9. Cai X, Lyu MR. The Effect of Code Coverage on Fault Detection Under Different Testing Profiles, *ICSE 2005 Workshop on Advances in Model-Based Software Testing (A-MOST)*, St. Louis, Missouri, May 2005.
10. Barnard G, Bayes T. Studies in the history of probability and statistics: IX. Thomas Bayes essay towards solving a problem in the doctrine of chances, *Biometrika*. 1985; 45: 293–315p.
11. Knafl GJ. Solving maximum likelihood equations for two-parameter software reliability models using grouped data, *Proceedings Third International Symposium on Software Reliability Engineering*, IEEE, 1992, 205–213p.
12. Minohara T, Tohma Y. Parameter estimation of hypergeometric distribution software reliability growth model by genetic algorithms, *International Symposium on Software Reliability Engineering*, IEEE. 1995, 324–329p.

Cite this Article

Bisma Gulzar Mir, Sheikh Riyaz-ul-Haq, Parameter Estimation of Software Reliability Growth Model. *Journal of Artificial Intelligence Research & Advances*. 2020; 7(1): 15–22p.