CHAPTER: 1

INTRODUCTION

Today's industry systems more and more depend on software which is sometimes very complex. Software complexity increases also with the risk factor of the environment where the whole system is deployed. From these reasons the requirements for software reliability cannot be missed out when designing such system. Software reliability differs from hardware reliability because it reflects the design perfection, rather than manufacturing perfection. Similar to hardware the reliability of software should be evaluated and measured, even it is not so simple task as it is in hardware because software is hard to touch. Many organizations and individuals developed methods for software reliability evaluation.

Prediction, progress, and process improvement. Measurement permeates everyday life and is an essential part in every scientific and engineering discipline. Measurement allows the acquisition of information that can be used for developing theories and models, and devising, assessing, and using methods and techniques. Software measurement is a way to track the process. As Grady states, "Without such measures for managing software, it is difficult for any organization to understand whether it is successful, and it is difficult to resist frequent changes of strategy". However, software engineering differs from other engineering disciplines in a number of aspects that have important consequences on software measurement. First, software engineering is a young discipline, so its theories, methods, models and techniques still need to be fully developed and assessed. However, the very nature of software engineering makes measurement a necessity, because more rigorous methods for production planning, monitoring,
and control are needed, otherwise the amount of risk of software projects may become excessive, and software production may easily get out of industrial control.

1.1 SOFTWARE ENGINEERING

Software engineering is an engineering discipline that is concerned with all aspects of software production. Software engineering is concerned with theories, methods and tools for professional software development. Software engineering is about creating high-quality software in a systematic, controlled, and efficient manner. Consequently, there is important emphasis on analysis and evaluation, specification, design and evolution of software. In addition, there are issues related to management and quality, to novelty and creativity, to standards, to individual skills, and to and professional practice that play a vital role in software engineering. In the historical development of computing, computer scientists produced software and electrical engineers produced the hardware on which the software runs. As the size, complexity, and critical importance of software grew, so did the need to ensure that software performs as intended. By the early 1970’s, it was apparent that proper software development practices required not only the underlying principles of computer science, they also required both the analytical and descriptive tools developed within computer science and the rigor that the engineering disciplines bring to the reliability and trustworthiness of the artifacts they engineered. The art of programming only is no longer sufficient to construct large programs. There are serious problems in the cost, timeliness, maintenance and quality of many software products. Software engineering has the objective of solving these problems by producing good quality, maintainable software, on time, within budget. To achieve this objective, we have to focus in a disciplined manner on both the quality of the product and on the process used to develop the
product. The IEEE Computer Society’s Software Engineering Body of Knowledge defines "software engineering" as the application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software, and the study of these approaches; that is, the application of engineering to software. It is the application of Engineering to software because it integrates significant mathematics, computer science and practices whose origins are in Engineering.

1.2 Software Development

Software engineering is an engineering discipline that is concerned with all aspects of software production. It is a systematic approach to the analysis, design, assessment, implementation, test, maintenance of software. Software engineering is concerned with theories, methods and tools for the software development. Software engineering covers not only the technical aspects of building software systems, but also management issues such as directing programming teams, scheduling and budgeting.

For the development of the software in the systemic way there are software development life cycle consist of different models like waterfall model, spiral model, iterative model and prototype models. The software development process transforms a user's needs into software, and this process can integrate a subset of the following phases.

Requirement phase: - The first main step in software development is gathering the requirements. The software requirements can change which depends on the software product which is going too developed. After analysis of the system requirement the next step is to analyze of the software requirements or system requirements. We can say that the feasibility study is the software requirements analysis. In Requirements gathering and analysis phase the development team communicate with customers and analyze the requires.
complete analysis on this area a .detailed document or report is prepared in this phase.

The detailed document and report can be like what project plan is or what the schedule of the project is. The cost of the project is estimated.

Design phase: - Design phase is the important phase of system development life cycle. In this phase the database design, the design of the architecture is chosen, functional specification design, low level design documents, high level design documents is done in this phase.

If a do designing properly it means that the well structured and analyzed design document is prepared it would reduce the time taken in the next phase that are implementation and testing phase.

Coding phase: - On the Basis of the design document the coding is done. The small modules of the software are prepared. All the modules are then combined together.

Testing phase: - A software which is not tested properly that may be not reliable or of bad quality. In this phase the developed system is tested properly and on the basis of testing the reports are prepared that the system is having bugs or errors in system.

There are different types of testing like unit testing, integration testing, system testing, white box testing, black box testing etc. In the integration testing we can do testing after combining the different modules. In the white box testing we can do internal and external testing. In the black box testing we can do only external testing. On depending the testing method the reports of the bugs are generated.

After the testing phase system again goes to development phase or designing phase for the correction of errors. In this again testing is done. This process (testing process) will continues until the software is found to be error free. There are different testing tools are available for easily doing testing.
Maintenance phase: - Maintenance is a process which includes all the activity which is done after the installation of software. Maintenance helps to keep the system operational.

There are two types of maintenance:
Adaptive maintenance
Corrective maintenance.
Maintenance is done on the existing software. In the development we create new software. Maintenance is a process of understanding the existing software. In the maintenance the developer or maintainer try to software which they to modify. Understanding the software not only involves the understanding the code but it is also the related to the documents. The output of each phase serves as the input to the next. The general activities in the software are specification, development, validation and evolution.

When the software is completely developed first of all it is checked that it is properly developed. It means that it is giving correct output. Software Testing is process of error detection, error correction. Mainly in the software testing we check that the software is error free there is not any error. Software testing is checking the software under controlled conditions to verify that it works as specified or as defined.

Software testing is used to detect errors and to validate that what has been specified in the software is what the user actually wanted. It means that it is meeting the user requirements. Software Testing contains following steps

Verification: - Verification is the checking or against the pre defined requirements. In this we check that are we building the system right.

Error Detection: Testing should intentionally attempt to make things go wrong to determine if things happen when they shouldn’t or things don’t happen when they should.
Validation: It is the process of checking that what has been specified is what the user actually wanted. In this we check that we are building the right system. The Software testing is the process of analyzing software to detect the differences between existing and required conditions that is defects/errors/bugs and to evaluate the features of the software item. The purpose of testing is verification, validation and error detection in order to find problems and the purpose of finding those problems is to get them fixed.

1.3 RELIABILITY THEORY

For engineering purposes, the reliability is defined as “The probability that a device will perform its intended function during a specified period of time under stated conditions”. [1] This may be expressed in mathematical way as

\[ R(t) = \int_t^\infty f(x)dx \quad (i) \]

where \( f(x) \) is the function representing the failure probability density and \( t \) represents the length of time (which is assumed to start from time zero). If we want better understand the reliability field, we have to focus on four aspects coming out of definition above:

1. Reliability is a probability. This means that failure occurs randomly, it can be a individual or recurring event. The incidence for failures varies in time according to the chosen probability function. Reliability engineering is then concerned with meeting the specified probability of success, at a specified statistical confidence level.

2. Reliability is predicated on "intended function". This means that this is generally understood as the mean operation without any failure.

3. Reliability applies to a specified period of time or other unit. This means that reliability of the system is guaranteed e.g. for a specified time, kilometres, cycles, etc.
4. Reliability is restricted to operation under stated conditions. This constraint is clear because it is not possible to design a system for unlimited conditions. The operating environment must be taken in focus when designing and testing the system.

1.4 SOFTWARE RELIABILITY

Reliability in the general engineering sense, is the probability. It gives component or system in a defined environment will operate correctly for a specified period of time. An important issue in developing such software systems is to produce high quality software system that satisfies user requirements. Software reliability models specify some reasonable form for this distribution, and are fitted to data from a software project. Once a model demonstrates a good fit to the available data, it can be used to determine the current reliability of the software, and predict the reliability of the software at future times.

Software reliability is often defined as —the probability of failure-free operation in a defined environment for a specified period of time. A failure is the departure of software behavior from user requirements. This dynamic phenomenon has to be distinguished from the static fault (or bug) in the software code, which causes the failure occurrence as soon as it is activated during program execution. Since software does not deprecate like hardware, the reliability of software stays constant over time if no changes are made to the code or to the environmental conditions including the user behavior. However, if each time after a failure has been experienced the underlying fault is detected and perfectly fixed, and then the reliability of software will increase with time.

Software reliability is about to define the stability or the life of software system with different properties. These properties include the trustfulness of software system, software cost, execution time, software stability etc. The aspects related
to these software system includes the probability of software faults, frequency of fault occurrence, criticality of fault, associated module with respective fault etc. In a software development process, the pre estimation of software reliability is required to deliver the software product. According to the required level of software quality estimation of software cost, development time is also estimated. There are number of quality measure that approves the software reliability [2]. Each stage of software life cycle itself takes some time quantum to deal with software reliability. Higher the software quality, lesser the software maintainability.

Various approaches can be used to improve the reliability of software, however, it is hard to balance development time and budget with software reliability. The main problem is that the software systems are complex so that software engineers are not currently able to test software well enough to insure its correct operation. Due to the assumptions made by various software reliability models, or due to there is dependence among successive software runs [3,4]. These problems are:

i) By finding mechanisms or relationships to more accurately determine the quality of software systems, without visiting a large fraction of their possible states.

ii) Taking in consideration the failure correlation and;

iii) Considering there is no single model sufficiently trustworthy in most or all applications.

There are many ways of using parametric models, nonlinear time series analysis and data mining to model software reliability and quality have been investigated. Fuzzy logic, neural networks, genetic algorithm, genetic programming and evolutionary computation are the most important key methodologies. Failure and
fault are two different factors which are generally inbuilt in our software during the development phase. Fault can be said as an error or bug which is introduced during the development phase. These investigations point the way towards using computational intelligence technologies to support human developers in creating software systems by exploiting the different forms of uncertainty present in a software system results from infrequent and unpredictable occurrence of human errors and incomplete or imprecise data, in order to model complex systems and support decision making in uncertain environments [5].

Software Reliability is an important attribute of software quality, functionality, usability, performance, serviceability, capability, install ability, maintainability, and documentation. It is hard to achieve, because the complexity of software tends to be high. There are three major classes of software reliability:

1) **Black box reliability analysis**: Estimation of the software reliability based on failure observations from testing or operation. These approaches are called black box approaches because internal details of the software are not considered.

2) **Software metric based reliability analysis**: Reliability evaluation based on the static analysis of the software (e.g., lines of code, number of statements, complexity) or its development process and conditions (e.g., developer experience, applied testing methods).

3) **Architecture-based reliability analysis**: Evaluation of the software system reliability from software component reliabilities and the system architecture (the way the system is composed out of the components). These approaches are sometimes called component-based reliability estimation (CBRE), or grey or white box approaches.
1.5 Software Defect:

It is impossible to produce defect-free software products, however, the main purpose of any software engineering activity is to prevent defects from being introduced in the first place.

A defect is defined as:

“An incorrect step, process, or data definition in a computer program.”

There is confusion in the terminology used concerning the terms defects, mistakes, errors and failures. The difference between the terms is explained by:

The fault tolerance discipline distinguishes between a human action (a mistake), its manifestation (a hardware or software fault), the result of the fault (a failure), and the amount by which the result is incorrect (the error).

Software defects are unique compared to physical defects in a product. They are harder to assess, and more cunning. Software does not deteriorate like physical products. The deterioration of software is a product of side effects from changes in the product in order to assess defects or changed requirements. This implies the total defect count may increase after release during maintenance. There is a risk of introducing new defects when changing the code.

Software defects can be injected during any phase of the software development life cycle. Jones lists the following as sources of defects: requirement errors, design defects, coding defects, documentation defects and incorrect corrections. A defect is injected when an
employee does a mistake in the work performed which creates a defect. A failure in the software might manifest itself and be discovered in either testing or inspections. However, not all defects are discovered before delivery. Such defects are latent defects and can be harder to locate, and advanced users are more likely to discover these than normal users.

Lyu proposes for techniques which is used in the software development life cycle in order to reduce the amount of defects:

- Defect prevention takes aim to reduce the number of defects introduce while producing the software in the software development life cycle. This is done in directly in any software engineering activity.

- Defect removal is to detect defects by software verification or software inspection. The main goal is to eliminate introduced defects. Strategies to achieve this may be dynamic analysis, or formal inspections of code.

- Defect tolerance is to provide continuous software service which satisfies given requirements despite a defect having occurred in the software.

- Defect forecasting is to estimate where new defects are likely to emerge in the software.

It is favourable to attain a high degree of defect removal efficiency in the organization according to Jones. The effects of this will have ramifications well beyond reducing the number of existing non-discovered defects in the software. The effects will allow the organisation to hit minimum schedules, maximize productivity, increase work environment satisfaction, fewer delivered defects, lower maintenance costs and lower risks of legal issues. The most effective way of improving software productivity is to
lower the number of defects in the software. This reduction can happen through defect prevention and defect removal, as described.

### 1.5.1 Impact Of Software Defects:

Many instances have tried to quantify the economic loss due to defects in software. Also, software defects impose negative social effects on both software users and software developers. Basili and Boehm assembled a list of ten rules of thumb which main purpose was to highlight pitfalls in software engineering. Seven of these rules are directly related to how defects impact the project. First, it is 100 times more expensive to correct a defect after delivery than during the requirements or design phase. Second, 40 to 50 percent of the effort in the project is spent on rework which could have been avoided. Third, 20 percent of the defects results in 80 percent of the rework and 80 percent of the defects come from 20 percent of the modules while half of modules are almost free of defects. Developing high dependability software are often 50 percent more expensive than low dependability software, and 90 percent of the downtime comes from 10 percent of the defects. However, investing the 50 percent extra is well worth it if the software is to be maintained. Last, 40 to 50 percent of software contains defects which are nontrivial. The results described by Boehm and Basili gives an impression of how software defects influence the total costs of software projects.

Defects only generate significant costs if they required redesign of the system, and the costs growth of delaying a correction of the defect till later phases supports Boehm and Basili. The type of typical defects introduced by each phase is distinct, and they have a varying degree of impact on budgets and schedule. Jones describes the defects introduced during the
requirements phase as the hardest to locate and correct. On the other hand, the defects introduced through the code are the most numerous, however, they are the easiest to locate and correct. The defects introduced during the design phase are the most grave, while defects in the documentation can be severe if ignored. Incorrect corrections of defects are very difficult to locate, and poorly designed test cases are often more of a burden than help. Another aspect is the data quality, which is hard to measure.

1.5.2 Software Process Improvement With Software Defects:

Software process improvement (SPI) is viewed as improving the software processes for the intent of increasing the quality of the software products. This can be done through understanding the original software process and change it in order to increase the quality of the software products. Grady claims software defect data is the most valuable source of information for software process improvement decisions. Further, the defect data provides a way of comparing improvements done against historic defect data in order to measure the effect of the improvements. He argues how ignoring defect data might yield dire consequences for business performance of an organization through reduced customer satisfaction and increased operational costs.

Grady claims software defect data is the most valuable source of information for software process improvement decisions. Further, the defect data provides a way of comparing improvements done against historic defect data in order to measure the effect of the improvements. He argues how ignoring defect data might yield dire consequences for business performance of an organization through reduced customer satisfaction and increased operational costs.
There are three ways organizations approach the handling of defects according to Basili and Fredericks.

1.6 SOFTWARE RELIABILITY GROWTH MODELS
Software reliability growth models, refers to those models that try to predict software reliability from test data [6]. These models try to show a relationship between fault detection data (i.e. test data) and known mathematical functions such as logarithmic or exponential functions. The goodness of fit of these models depends on the degree of correlation between the test data and the mathematical function [7].

The NHPP models are the most popular ones. The reason is the NHPP model has ability to describe the software failure phenomenon. The first NHPP model, which strongly influences the development of many other models presented a NHPP model with S-shaped mean value function. They [10, 11] also made further progress in various S-Shaped NHPP models. Although these NHPP models are widely used, they impose certain restrictions or a priori assumptions about the nature of software faults and the stochastic behavior of software failure process.

1.7 SOFTWARE RELIABILITY GROWTH MODELS (SRGMS) AND CRITERIA
A software reliability growth model (abbreviated as SRGM) is known as one of the fundamental technologies for quantitative software reliability assessment, and playing an important role in software project management for producing a highly-reliable software system[12]. SRGM is mathematical model, shows how software reliability improves as faults are detected and repaired. SRGM can be used to predict when a particular level of reliability is likely to be attained. Thus, SRGM is used to determine when to stop testing to attain a given reliability level [13]. There are many software reliability growth models but the commonly used model of software reliability models are JM, GO model, MO model, Sch model, S-Shape model. To evaluate the prediction powers of different models, it is necessary to use a meaningful measures. They use two criteria: Root Mean Square Error (RMSE) and Average Error(AE). These criteria are used to measure the difference between the actual and predicted values, the formulas is as Equation (1) (2).

\[ RMSE = \sqrt{\frac{1}{n} \sum_{i=1}^{n} (c(k) - \hat{c}(k))^2} \]  
\[ AE = \frac{1}{n} \sum_{i=1}^{n} \left| \frac{c(k) - \hat{c}(k)}{c(k)} \right| \times 100 \]  

Where n is the number of groups of failure data, c(k) is the number of the actual failures in each group of failure data, \( \hat{c}(k) \) is the number of the predicted failures. The smaller the RMSE and AE, the stronger that the model prediction ability [6].

1.8 Various types of Models

1.8.1 Jelinski-Moranda Model

The Jelinski-Moranda model was first introduced as a software reliability growth model in Jelinski and Moranda (1972) [14]. This is a continuous time-
independently distributed inter failure times and independent and identical error behavior model [15]. The software failure rate of hazard function at any time is proportional to the current fault content of the program. The distribution of the order statistics is the Exponential distribution [14].

The main assumptions for the Jelinski-Moranda model are the following:

1. At the beginning of testing, there are \( n_0 \) faults in the software code with \( n_0 \) being an unknown but fixed number.
2. All faults are of the same type.
3. Immediate and perfect repair of faults.
4. Faults are detected independently of each other.
5. The times between failures are exponentially distributed with parameter proportional to the number of remaining faults.
6. Each fault is equally dangerous with respect to the probability of its instantaneously causing a failure. Furthermore, the hazard rate of each fault does not change over time, but remains constant at \( \varphi \).
7. The failures are not correlated, i.e. given \( n_0 \) and \( \varphi \) the times between failures \( (\Delta t_1, \Delta t_2, \ldots, \Delta t_{n_0}) \).
8. Whenever a failure has occurred, the fault that caused it is removed instantaneously and without introducing any new fault into the software.

\[ z(\Delta t | t_{i-1}) = \phi[n_0 - M(t_{i-1})] = \phi[n_0 - (i - 1)] \]  

(1)

The failure intensity function is the product of the inherent number of faults and the probability density of the time until activation of a single fault, \( n_a(t) \), i.e.:

\[ \frac{d n(t)}{d(t)} = n_0[1 - \exp(-\varphi t)] \]  

(2)

Therefore, the mean value function is

\[ m(t) = n_0[1 - \exp(-\varphi t)] \]  

(3)
It can easily be seen from equations (2) and (3) that the failure intensity can also be expressed as

\[
\frac{dm(t)}{d(t)} = \phi[n_0 - m(t)]
\]

According to equation (4), the failure intensity of the software at time \( t \) is proportional to the expected number of faults remaining in the software; again, the hazard rate of \( n \) individual faults is the constant of proportionality. Moreover, many software reliability growth models can be expressed in a form corresponding to equation (4).

One of the most widely discussed assumptions of the Jelinski-Moranda model is (2) since it implies that each repaired fault reduces the hazard rate of the new time between failure by a constant \( \lambda > 0 \). This idea is depicted in Figure 1.1.

![Diagram of Jelinski-Moranda model hazard rate](image)

**Figure 1.1: Jelinski-Moranda model hazard rate. It remains constant between failure observations and decreases by a factor \( \lambda \) after a fault is repaired [14]**

### 1.8.2. Goel-Okumoto Model

This model, first proposed by Goel and Okumoto, is one of the most popular NHPP model in the field of software reliability modeling. It is also called the exponential NHPP model. Assumptions (2), (3) and (4) for the Jelinski-Moranda model are also valid for the Goel-Okumoto model. Considering failure detection
as a Non homogeneous Poisson process with an exponentially decaying rate function, the mean value function is hypothesized in this model as [16]

\[ m(t) = a(1 - \exp[-bt]), \quad a > 0, b > 0 \]

and the intensity function of this model is given as

\[ \lambda(t) = ab \times \exp[-bt], \quad a > 0, b > 0 \]

where a is the expected total number of faults to be eventually detected and \( b \) represents the fault detection rate.

In fact, it follows that [14]

\[ \lim_{t \to \infty} m(t) = a \]

A typical plot of \( m(t) \) for the Goel-Okumoto model can be observed in Figure 1.2 where \( m(t) \) is plotted when \( a = 11 \) and \( b = 0.14 \). Note that \( a \) determine the scale and \( b \) the shape of the mean-value function.

Figure 1.2: Goel-Okumoto model mean-value function when \( a = 11 \) and \( b = 0.14 \) [14]
1.8.3 Generalized Goel NHPP Model

In order to describe the situation that software failure intensity increases slightly at the beginning and then begins to decrease, Goel proposed a simple generalization of the Goel-Okumoto model with an additional parameter $c$ [16]. The mean value function and intensity function are

\[ m(t) = a(1 - \exp[-bt^c]), \quad a > 0, b > 0, c > 0 \]

\[ \lambda(t) = abc t^{c-1} \exp[-bt^c], \quad a > 0, b > 0, c > 0 \]

where $a$ is the expected total number of faults to be eventually detected and $b$ and $c$ are parameters that reflect the quality of testing.

1.8.4 Inflected S-Shaped Model

This model solves a technical problem in the Goel-Okumoto model. It was proposed by Ohba and its underlying concept is that the observed software reliability growth becomes S-shaped if faults in a program are mutually dependent, i.e., some faults are not detectable before some others are removed. The mean value function is [16]

\[ m(t) = a * \frac{1 - \exp[-bt]}{1 + \psi(r) * \exp[-bt]}, \quad \psi(r) = \frac{1 - r}{r}, \quad a > 0, b > 0, r > 0 \]

The parameter $r$ is the inflection rate that indicates the ratio of the number of detectable faults to the total number of faults in the software, $a$ is the expected total number of faults to be eventually detected, $b$ is the fault detection rate, and $r$ is the inflection factor. On taking \( \psi(r) = \beta \) then the inflection S-shaped model mean value function and intensity function are given as

\[ m(t) = a * \frac{1 - \exp[-bt]}{1 + \beta * \exp[-bt]}, \quad a > 0, b > 0, \beta > 0 \]
1.8.5 Logistic Growth Curve Model

In general, software reliability tends to improve and it can be treated as a growth process during the testing phase. That is, the reliability growth occurs due to fixing faults. Therefore, under some conditions, the models developed to predict economic population growth could also be applied to predict software reliability growth. These models simply fit the cumulative number of detected faults at a given time with a function of known form. Logistic growth curve model is one of them and it has an S-shaped curve. Its mean value function and intensity function are [16]

\[ m(t) = \frac{a}{1 + k \exp[-bt]} , \quad a > 0, b > 0, k > 0 \]

\[ \lambda(t) = \frac{ab\exp[-bt]}{(1 + k \exp[-bt])^2} , \quad a > 0, b > 0, k > 0 \]

where \( a \) is the expected total number of faults to be eventually detected and \( k \) and \( b \) are parameters which can be estimated by fitting the failure data.

1.8.6 Musa-Okumoto Model

Musa-Okumoto have been observed that the reduction in failure rate resulting from repair action following early failures are often greater because they tend to the most frequently occurring once, and this property has been incorporated in the model [16]. The mean value function and intensity function of the model given as

\[ m(t) = a \ln(1 + bt) , \quad a > 0, b > 0 \]

\[ \lambda(t) = \frac{ab}{(1 + bt)} , \quad a > 0, b > 0 \]
where \( a \) is the expected total number of faults to be eventually detected and \( b \) is the fault detection rate.

**TABLE I: MEAN VALUE AND INTENSITY OF VARIOUS MODELS**

<table>
<thead>
<tr>
<th>Model Name</th>
<th>Mean Value Function</th>
<th>Intensity Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Jelinski-Moranda model</td>
<td>( m(t) = n_0[1 - \exp(-\phi t)] )</td>
<td>( \lambda_i = (N - k)\mu )</td>
</tr>
<tr>
<td>2. Goel-Okumoto Model</td>
<td>( m(t) = a(1 - \exp[-bt]), a &gt; 0, b &gt; 0 )</td>
<td>( \lambda(t) = ab \exp[-bt], a &gt; 0, b &gt; 0 )</td>
</tr>
<tr>
<td>3. Generalized Goel NHPP Model</td>
<td>( m(t) = a(1 - \exp[-bt^c]), a &gt; 0, b &gt; 0, c &gt; 0 )</td>
<td>( \lambda(t) = abct^{c-1} \exp[-bt^c], a &gt; 0, b &gt; 0, c &gt; 0 )</td>
</tr>
<tr>
<td>4. Inflected S-Shaped Model</td>
<td>( m(t) = a \frac{1 - \exp[-bt]}{1 + \psi(r) \exp[-bt]}, \psi(r) = \frac{1-r}{r}, a &gt; 0, b &gt; 0, r &gt; 0 )</td>
<td>( \lambda(t) = \frac{ab \exp[-bt](1 + \beta t)}{(1 + \beta \exp[-bt])^2}, a &gt; 0, b &gt; 0, \beta &gt; 0 )</td>
</tr>
<tr>
<td>5. Logistic Growth Curve Model</td>
<td>( m(t) = \frac{a}{1 + k \exp[-bt]}, a &gt; 0, b &gt; 0, k &gt; 0 )</td>
<td>( \lambda(t) = \frac{ab \exp[-bt]}{(1 + k \exp[-bt])^2}, a &gt; 0, b &gt; 0, k &gt; 0 )</td>
</tr>
<tr>
<td>6. Musa-Okumoto Model</td>
<td>( m(t) = a \ln(1 + bt), a &gt; 0, b &gt; 0 )</td>
<td>( \lambda(t) = \frac{ab}{1 + bt}, a &gt; 0, b &gt; 0 )</td>
</tr>
</tbody>
</table>
1.9 NEURAL NETWORK

Neural networks are a computational metaphor inspired studies of the brain and nervous system in biological organisms. They are highly idealized mathematical models of how understand the essence of these simple nervous systems. The basic characteristics of a neural network [17] are

- It consists of many simple processing units, called neurons that perform a local computation on their input to produce an output.
- Many weighted neuron interconnections encode the knowledge of the network.
- The network has a learning algorithm that lets it automatically develop internal representations.

One of the most widely used processing unit models is based on the logistic function. The resulting transfer function is given by

\[
output = \frac{1}{1 + e^{-sum}}
\]

Where sum is the aggregate of weighted inputs.

Figure 1.3 shows the actual I/O response of this unit model. The unit is nonlinear and continuous. There exists a variety of neural network models and learning procedures. Two well-known classes of neural networks that can be
used for prediction applications are: feed-forward networks and recurrent networks. They use feed-forward networks and learning procedure is back propagation algorithm which comes under the category of supervised learning.

1.9.1 Architecture Of Neural Network
The artificial neural network is designed on the basis of the study of biological neural network. A human brain is responsible for all the action and reaction taken by human body. These actions are controlled by brain and are carried to different parts of the body by a fast processing node known as neurons. Similarly in case of artificial neurons also we have a fast processing node responsible for performing the computation and are known as Neurons [18]. A neural network is a collection one to multiple neurons which are arranged in a specific manner to perform the computation. The basic structure of a neuron is represented in the given below figure1.4:

![Structure of a Neuron](image)

**Figure 1.4: Structure of a Neuron**

The model is designed for a set of input values and corresponding desired output. Suppose consider that „I” is the set of input values such that I={i1,i2……in} and
we associate a variable parameter with it which controls the behavior of neurons. Let us consider weights associated with the input as the variable parameter such that \( W = \{ w_1, w_2, \ldots, w_n \} \). All the input along with the weight is passed through the summation unit where compute the net input which given by

\[
\text{Netinput} = \sum_{i=1}^{n} i_i w_i \text{ where } n = \text{number of input to the network} \ldots \ldots \\
= i_1 w_1 + i_2 w_2 + i_3 w_3 + \ldots \ldots + i_n w_n
\]

These neurons are arranged in different layers to represent a specific structure known as neural network architecture.

![Multi Layer Neural Networks](image)

*Figure 1.5: Multi Layer Neural Networks*

For designing a reliability model we will require a multilayer of neurons. The neurons are arranged in three different layers known as input layer, hidden layer and output layer. The multi layer network is shown in below Figure 1.5.

### 1.9.2 ROLE OF NEURAL NETWORK IN SOFTWARE RELIABILITY

The reliability model till now been designed are based on the study of failure associated with the code and the environment where it is been implemented [19]. All the software reliability models are designed on the basis of execution time and calendar time. Execution time of any program is the time that is actually
required or spent by the processor in executing the instruction of that program [20]. Calendar time is referred as the elapsed time from start to end of program execution on a running computer.

Neural network is a collection of fast processing and computing nodes called artificial neurons. These neurons are designed on the basis of study of the behavior of biological neuron. These neurons are connected in a specific manner that is layer like structure known as neural network architecture. Neural network is an output based computing technique. It is a technology generally been used for optimizing problem. Neural networks also consist of multiple layers of computational units, usually interconnected in a feed-forward way. Neural network [21] has been applied to estimate parameters of the formal model and to learn the process itself in order to predict the future outcomes. It has been shown that feed forward network can be applied for prediction. Back-error propagation is one of the most widely used neural network paradigms and has been applied successfully in application studies in a broad range of areas [22].

Each neuron in one layer has directed connections to the neurons of the subsequent layer. In many applications the units of these networks apply a sigmoid function as an activation function. ANN (Artificial Neural Network) software reliability models have recently aroused more research interest [23, 24, 25, 26]. Traditionally, both kinds of models only consider single fault detection process (FDP) and data for analysis are only from FDP. However, while data from both FDP&FCP (fault correction process) are available, NHPP and ANN models can be extended into paired NHPP models and combined ANN models, providing more accurate predictions [23]. Generally speaking, data-driven approach is much less restrictive in assumptions compared to analytical approach.