Parameter Estimation

A software reliability model is a function of various parameters. Fitting this function to the data means estimating its parameters from the data. There are two techniques for parameter estimation:

1. One approach to estimating parameters is to input the data directly into equations for the parameters.
    - The most common method for this direct parameter estimation is the maximum likelihood technique .
2. A second approach is fitting the curve described by the function to the data and estimating the parameters from the best fit to the curve.
    - The most common method for this indirect parameter estimation is the least squares technique.

Maximum Likelihood Technique

- The maximum likelihood technique consists of solving a set of simultaneous equations for parameter values. The equations define parameter values that maximize the likelihood that the observed data came from a distribution with those parameter values.
- Maximum likelihood estimation satisfies a number of important statistical conditions for an optimal estimator and is generally considered to be the best statistical estimator for large sample sizes**.**
- Unfortunately, the set of simultaneous equations it defines are very complex and usually have to be solved numerically. Classical Least Squares
- The maximum likelihood technique solves directly for the optimal parameter values.

Least Squares Technique

- The least squares method solves for parameter values by picking the values that best fit a curve to the data.
- This technique is generally considered to be the best for small to medium sample sizes.
- The theory of least squares curve-fitting is that we want to find parameter values that minimize the "difference" between the data and the function fitting the data
    - the difference is defined as the sum of the squared errors.
    - The classical least squares technique involves log likelihood functions

Musa Basic Execution Time Model versus Logarithmic Poisson Model

The Musa basic execution time model assumes that all faults are equally likely to occur, are independent of each other and are actually observed. The execution times between failures are modelled as piecewise exponentially distributed. The intensity function is proportional to the number of faults remaining in the program and the fault correction rate is proportional to the failure occurrence rate.

Logarithmic Poisson execution time model, also called the Musa-Okumoto model, also assumes that all faults are equally likely to occur and are independent of each other. The expected number of faults is a logarithmic function of time in this model, and the failure intensity decreases exponentially with the expected failures experienced. Finally, the software will experience an infinite number of failures in infinite time.

Both the basic and logarithmic Poisson execution time models are based on the premise that execution time (the actual processor time used in executing the program) is the best time domain for expressing reliability. Execution time is the most practical measure of the failure-inducing stress being placed on a program. Both models consist of two components: an execution time component and a calendar time component. The former component characterizes reliability behavior as a function of execution time r. The latter component relates execution time r to calendar time t, which is more useful for managers and engineers in expressing when a specified reliability goal is expected to be reached.

Comparison criteria

The comparison criteria that are described below represent an approximate consensus among a number of researchers in the field [8]. They are:

a) predictive validity
b) capability
c) quality of assumptions
d) applicability
e) simplicity.

Predictive Validity

It is the ability of the model to determine future failure behavior during either the test or the operational phases from present and past failure behavior in the respective phase. In comparing predictive validity of models, one must keep in mind that differences should be larger than other sources of error (especially measured error) before any advantage can be attributed to one model over another.

Capability

Capability refers to the ability of the model to estimate with satisfactory accuracy quantities needed by software managers, engineers, and users in planning and managing software development projects or controlling change in operational software systems. The degree of capability must be gauged by looking at the relative importance as well as number of quantities estimated. The quantities, in approximate order of importance are:

I. present reliability, MTTF, or failure intensity
II. expected date of reaching a specified reliability, MTTF, or failure intensity goal (it is assumed that the goal is variable and that dates can be computed for a number of goals, if desired. If a date cannot be computed and the goal achievement can be described only in terms of additional execution time or failures experienced, this limited facility is preferable to no facility, although it is very definitely inferior
III. resource and cost requirements related to achievement of the foregoing goal(s).

Any capability of a model for prediction of software reliability in the system design and early development phases would be extremely valuable because of the resultant value for system engineering and planning purposes. These predictions must be made through measurable characteristics of the software (size, complexity, structure, etc.), the software development environment and the operational environment.

Quality of Assumptions

The following considerations of quality should be applied to each assumption in turn:

- If it is possible to test an assumption, the degree to which it is supported by actual data is an important consideration. This is especially true of assumptions that may be common to an entire class of models.
- If it is not possible to test the assumption, its plausibility from the viewpoint of logical consistency and software engineering experience should be evaluated.
- Finally, the clarity and explicitness of an assumption should be judged; these characteristics are often necessary to determine whether a model applies to particular circumstances.

Applicability

Another important characteristic of a model is its applicability. A model should be judged on its degree of applicability across different software products (size, structure, function, etc.), different development environments, different operational environments, and different life cycle phases. However, if a particular model gives outstanding results for just a narrow range of products or development environments, it should not necessarily be eliminated. There are at least five situations that are encountered commonly enough in practice that a model should either be capable of dealing with them directly or should be compatible with procedures that can deal with them. There are:

I. phased integration of a program during test (i.e., testing starts before the entire program is integrated, with the result that some failure data is associated with a partial program)
II. design and requirements changes to the program
III. classification of severity categories
IV. ability to handle incomplete of failures into different failure data or data with measurement uncertainties (although not without loss of predictive validity)

V. operation of same program on computers of different performance. Finally, it is desirable that a model be robust with respect to departures from its assumptions, errors in the data or parameters it employs, and unusual conditions.

Simplicity

A model should be simple in three aspects:

I. The most important consideration is that it must be simple and inexpensive to collect the data that is required to particularize the model. If the foregoing is not the case, the model will not be used.

II. Second, the model should be simple in concept. Software engineers without extensive mathematical background should be able to understand the nature of the model and its assumptions, so they can determine when it is applicable and the extent to 85 which the model may diverge from reality in an application. Parameters should have readily understood interpretations that relate to characteristics of the program, the development environment, or the execution environment. This property makes it more feasible for software engineers to estimate the values of the parameters where data is not available.

III. Finally, a model must be readily implementable as a program that is a practical management and engineering tool. This means that the program must run rapidly and inexpensively with no manual intervention required other than the initial input.

Comparison of Software Reliability Growth Models

Comparison of Different Software Reliability Models

The large number of proposed models tends to give quite different predictions for the same set of failure data. It should be noted that this kind of behavior is not unique to software reliability modeling but is typical of models that are used to project values in time and not merely represent current values. Even a particular model may give reasonable predictions on one set of failure data and unreasonable predictions on another. All this has left potential users confused and adrift with little guidance as to which models may be best for their application.

The search for the best software reliability model(s) started in the late 1970s and early 1980s and still continues today. Initial efforts at comparison by Schick and Wolverton (1978) and Sukert (1979) suffered from a lack of good failure data and a lack of agreement on the criteria to be used in making the comparisons.

The former deficiency was remedied to some degree in 1979 when over 20 reasonably good-quality sets of failure data were published (Musa 1979). The data sets were collected under careful supervision and control and represent a wide variety of applications including real time command and control,

commercial, military, and space systems and they ranged in size from about 6K object instructions to 2.4M object instructions.
The latter deficiency was remedied when Iannino and co-workers (1984) worked out a consensus from many experts in the field on the comparison criteria to be employed. The proposed criteria include the following.

1. The capability of a model to predict future failure behavior from known or assumed characteristics of the software; for example, estimated lines of code, language planned to be used, and present and past failure behavior (that is, failure data). This is significant principally when the failure behavior is changing, as occurs during system testing.
2. The ability of a model to estimate with satisfactory accuracy the quantities needed for planning and managing software development projects or for running operational software systems. These quantities include the present failure intensity, the expected date of reaching the failure intensity objective, and resource and cost requirements related to achieving the failure intensity objective.
3. The quality of modeling assumptions (for example, support by data, plausibility, clarity, and explicitness).
4. The degree of model applicability across software products that vary in size, structure, and function, different development environments, different operational environments, and different life cycle phases. Common situations encountered in practice that must be dealt with include programs being integrated in phases, system testing driven by a strategy to test one system feature at a time, the use of varying performance computers, and the need to handle different failure severity classes.
5. The degree of model simplicity (for example, simple and inexpensive data collection, concepts, and computer implementation).

The Software Reliability Growth Models are broadly classified based on the following dimensions:

- **Category** : Number of Failures is infinite or finite
- **Type** : Distribution of the number of Failures experienced by the time specified.. Two important types that can be consider are the Poisson and Binomial.
- **Class** (only Finite Category): Functional form of the failure intensity over time.
- **Time Domain**: Calendar or Execution time.
- **Family** (only Infinite Category):) Functional form of the failure intensity function expressed in terms of the expected number of failures experienced.

Some of the models based on different classification can be briefly discussed as follows:

Musa -Basic Model:

Assumptions:

- The detections of failures are independent of one another.

- The software failures are observed(i.e, the total number of failures has an upper bound)
- The execution times (measure in CPU time) between failures are piecewise exponentially distributed.
- The hazard rate is proportional to the number of faults remaining in the program.
- The fault correction rate is proportional to the failure occurrence rate.

Non Homogeneous Poisson Process Model:

Assumptions:

- The cumulative number of failures detected at any time follows a Poisson distribution.
- Time periods (intervals) can be unequal.
- Perfect debugging is assumed.

Data requirements:

- Fault counts on each testing interval f1,f2,…fn
- Completion time of each period t1,t2,….tn

Geometric Model:

The time between failures has an exponential distribution whose mean decreases in geometric fashion (i.e, the earlier faults have larger impact)

Assumptions:

- There is no upper bound on the total number of failures.
- All faults do not have the same chance of detection.
- The detections of faults are independent of one another.
- The failure detection rate forms a geometric progression and is constant between failure occurrence

Data requirements:

- Either actual failure time
- Or Elapsed time between failure