Defect / Effort Prediction Models in Software Maintenance Projects

Mr. S. Manivannan¹, Dr. S. Balasubramanian²

ABSTRACT

Planning software maintenance work is a key factor for a successful maintenance project and for better project scheduling, monitoring, and control. To this aim, effort estimation is a valuable asset to maintenance managers in planning maintenance activities and performing cost/benefits analysis. Software practitioners recognize the importance of realistic estimates of effort to the successful management of software projects. Having realistic estimates at an early stage in a project's life cycle allow project managers and software organizations to manage resources effectively. Prediction is a necessary part of an effective process, be it authoring, design, testing, or Web development as a whole. Maintenance projects may range from ordinary projects requiring simple activities of understanding, impact analysis and modifications, to extraordinary projects requiring complex interventions such as encapsulation, reuse, reengineering, migration, and retirement. Moreover, software costs are the result of a large number of parameters so any estimation or control technique must reflect a large number of complex and dynamic factors. This paper mainly covers the models related to Fault predictions well ahead of the execution so that it gives an insight to the managers to take appropriate decisions.

¹ Research Scholar, Anna University Coimbatore, Coimbatore – 641 046
Email: smanivannan2003@yahoo.com

² IPR Consultant & Research Supervisor, Anna University Coimbatore, Coimbatore – 641 046
Email: s_balasubramanian@rediffmail.com
Data about the defects found during software testing is recorded in software defect reports or bug reports. The data consists of defect information including defect number at various testing stages, complexity (of the defect), severity, information about the component to which the defect belongs, tester, and person fixing the defect. Reliability models mainly use data about the number of defects and its corresponding time to predict the remaining number of defects. This research work proposes an empirical approach to systematically elucidate useful information from software defect reports by (1) employing a data exploration technique that analyzes relationships between software quality of different releases using appropriate statistics, and (2) constructing predictive models for forecasting time for fixing defects using existing machine learning and data mining techniques.

I. INTRODUCTION

A "field fault" is a software fault that goes undetected in the development debugging and testing, and is identified in the customer's operational system. A "risk model" is a formula or set of rules to predict whether a software module is more likely to have a field fault. A "module" refers to the component of code that is managed by the CM system. It is the entity at which faults are tracked and it is the target of the software risk model. A "release" is a version of the entire software package that is delivered to a customer. If it is not released to the customer, then there is no field fault data.
Figure 1: Software development activities

The activities of the software development process represented in the waterfall model in Figure 1. There are several other models to represent this process.

Planning

The important task in creating a software product is extracting the requirements or requirements analysis. Customers typically have an abstract idea of what they want as an end result, but not what software should do. Incomplete, ambiguous, or even contradictory requirements are recognized by skilled and experienced software engineers at this point. Frequently demonstrating live code may help reduce the risk that the requirements are incorrect. Once the general requirements are gleaned from the client, an analysis of the scope of the development should be determined and clearly stated. This is often called a scope document. Certain functionality may be out of scope of the project as a function of cost or as a result of unclear requirements at the start of development. If the development is done externally, this document can be considered a legal document so that if there are ever disputes, any ambiguity of what was promised to the client can be clarified.
Design

Domain Analysis is often the first step in attempting to design a new piece of software, whether it be an addition to an existing software, a new application, a new subsystem or a whole new system. Assuming that the developers (including the analysts) are not sufficiently knowledgeable in the subject area of the new software, the first task is to investigate the so-called "domain" of the software. The more knowledgeable they are about the domain already, the less work required. Another objective of this work is to make the analysts, who will later try to elicit and gather the requirements from the area experts, speak with them in the domain's own terminology, facilitating a better understanding of what is being said by these experts. If the analyst does not use the proper terminology it is likely that they will not be taken seriously, thus this phase is an important prelude to extracting and gathering the requirements. If an analyst hasn't done the appropriate work confusion may ensue: "I know you believe you understood what you think I said, but I am not sure you realize what you heard is not what I meant.

Architecture

The architecture of a software system or software architecture refers to an abstract representation of that system. Architecture is concerned with making sure the software system will meet the requirements of the product, as well as ensuring that future requirements can be addressed. The architecture step also addresses interfaces between the software system and other software products, as well as the underlying hardware or the host operating system.

Implementation, testing and documenting

Implementation is the part of the process where software engineers actually program the code for the project. Software testing is an integral and important part
of the software development process. This part of the process ensures that bugs are recognized as early as possible. Documenting the internal design of software for the purpose of future maintenance and enhancement is done throughout development. This may also include the authoring of an API, be it external or internal.

**Deployment and Maintenance**

Deployment starts after the code is appropriately tested, is approved for release and sold or otherwise distributed into a production environment. Software Training and Support is important because a large percentage of software projects fail because the developers fail to realize that it doesn't matter how much time and planning a development team puts into creating software if nobody in an organization ends up using it. People are often resistant to change and avoid venturing into an unfamiliar area, so as a part of the deployment phase, it is very important to have training classes for new clients of your software.

Maintenance and enhancing software to cope with newly discovered problems or new requirements can take far more time than the initial development of the software. It may be necessary to add code that does not fit the original design to correct an unforeseen problem or it may be that a customer is requesting more functionality and code can be added to accommodate their requests. It is during this phase that customer calls come in and you see whether your testing was extensive enough to uncover the problems before customers do. If the labor cost of the maintenance phase exceeds 25% of the prior-phases' labor cost, then it is likely that the overall quality, of at least one prior phase, is poor. In that case, management should consider the option of rebuilding the system (or portions) before maintenance cost is out of control. Bug Tracking System tools are often deployed at this stage of the process to allow development teams to interface with customer/field teams testing the software to identify any real or perceived issues.
These software tools, both open source and commercially licensed, provide a customizable process to acquire, review, acknowledge, and respond to reported issues.

II. DEFECT PREDICTION / ANALYSIS

We are going to describe a two-step approach for analyzing software defect data. Like any data analysis technique, the first natural step is to explore the data by examining relevant statistics that can help us gain useful information. Many possible statistics exist but an important issue is how to find the appropriate ones. Techniques that can lead to the answer are likely to be domain dependent. On the other hand, instead of restricting the analysis on certain statistics, our second step expands data exploration to utilize the available data as much as possible by means of data mining techniques. There are many existing data mining algorithms and yet, most that have been applied to analyze defect data deal with only one or two types of problems (e.g., classification of software modules, prediction of number of defects or failures). The goal of our empirical study is to examine other types of problems whose empirical solutions provide potential benefits for software development practices. Our additional goal is to form a principled methodology in defect data analysis as opposed to creating algorithms and obtaining results. This chapter gives generic steps for defect data analysis and proposes a new type of problem to be investigated. Section below discusses the first step and the next section describes the second step of our proposed approach.

Analysis with Appropriate Statistics

Software defect data can be used to infer useful information about the quality of releases. Specifically, what is the quality of pre release and post release software given number of defects and components information? We will define some terms
and basic measures to help understand quality of pre release and post release software.

DBR - Defects Before Release,

DAR - Defects After Release,

N - Number of Modules (components, henceforth in this section when we say module we mean component from our data). These statistics or terms can be used in the following ways to infer useful information.

1. The ratio of DBR/N can be an indicator of the quality of the release. (when the ratio is compared to prior release ratio).

2. The ratio DAR/N, can be another indicator of the quality of the release. Between DAR/N and DBR/N, DAR/N is more significant as it is the quality which the users directly experience. But DAR cannot be known before the release (it may be useful for post release analysis).

The ratio of DBR/DAR can be an indicator of the efficiency of testing and defect removal process. In the ideal case there should be no defects after release in which case the ratio approaches infinity. Thus a high value of this ratio may be another indicator of desirable quality. A counter example would be when the ratio (DBR/DAR) is high but the actual number of defects is small. For example in release 1 if DBR = 100 and DAR = 25, DBR/DAR = 4. In release 2 DBR = 10 and DAR = 5, DBR/DAR = 2. Though the ratio is higher for release 1, we cannot say for sure that release 1 has better quality as the number of defects is significantly higher. To overcome this problem we first looked at the value of DBR across releases in step 1.
If you have a large number of defects before release, there can be two answers. One is that, because we have found and fixed all the defects we can expect the defects in post release software to be low. The other would be that, a lot of defects before release implies, the software or the components were buggy and hence we should expect more defects after release. The scenarios are given below:

1. Consider current release as release n. Divide modules in release n-1 into groups based on number of defects in release n-2. For example the first group of modules in release n-1 should only contain those modules that had no defects in release n-2. The second group should contain those modules, which had only one defect in release n-2 and so on. After we divide the modules into groups in the above manner, we can plot for each group the number of defects found before release in release n-1 against the average defects found before release in release n. This graph is similar to the graph we plotted in the above step except that the two sets of defects belong to different releases. But here the emphasis is not on individual graphs but on whether they differ from each other in an expected way or not. If the graphs for all the groups are similar then release n-2 has no effect on release n. If they show a consistent variation (example, if average height increases from groups 1 to 4, it indicates that release n-2 has some impact on release n.

Higher number of defects being found before release does not necessarily imply that fewer defects will be found after release. To investigate this divide the modules into groups based on number of defects (for example, group1 would have modules with 0 defects, group 2 would have modules with 1 defects, group 3 would have modules with 3 defects and so on). For each group of modules plot a graph of average defects after release against average defects before release. If the graph has positive slope it implies that defects after release increase with defects before release. On the other hand if it has negative slope it indicates defects after
release decrease as defects before release increases. This may indicate that a lot of
time was spent on testing and fixing bugs and that it was effective.

2. To determine if a previous release affects the quality of the current release, draw
graphs in the following way.
Consider current release as release n. Divide modules in release n-1 into groups
based on number of defects in release n-2. For example the first group of modules
in release n-1 should only contain those modules that had no defects in release n-2.
The second group should contain those modules, which had only one defect in
release n-2 and so on. After we divide the modules into groups in the above
manner, we can plot for each group the number of defects found before release in
release n-1 against the average defects found before release in release n. This
graph is similar to the graph we plotted in the above step except that the two sets
of defects belong to different releases. But here the emphasis is not on individual
tgraphs but on whether they differ from each other in an expected way or not. If the
graphs for all the groups are similar then release n-2 has no effect on release n. If
they show a consistent variation (example, if average height increases from groups
1 to 4, it indicates that release n-2 has some impact on release n. [Graph can also
be plotted for defects found in release n-1 against no of modules that are defects
free in release n. The same conclusions can be drawn from this graph too.]

This analysis can show us whether problems from release n-2 are affecting release
n or not. If release n-2 has no effect on release n, the developers can concentrate
more on features newly introduced in release n-1.
III. CONSTRUCTING PREDICTIVE MODELS FROM A DATASET

Decision Tree

A decision tree is a flow like structure, where each internal node denotes a test on an attribute, each branch represents an outcome of the test, and leaf nodes represent classes or class distributions. The top node is called the root node and the nodes on the bottom most level are called leaves. In order to classify an unknown sample the attributes are compared against the decision tree and a path is traced. The leaf node of the path gives the class of the sample. The branches can be thought of as representing conjunctions of conditions that lead to those classifications.

Algorithm for generating decision tree

1. Start with a node N in the tree.
2. If all the data points are of the same class then
3. Label node N with that class and stop.
4. If there are no attributes in the data (data has one attribute which is the class),
5. Return N labeled with the majority class and stop.
6. Select the attribute with the highest information gain. Let us call it a.
7. Label node N with the attribute a (selected in step 4).
8. For each known value v of attribute a
9. Develop a branch for the condition that a = v.
10. Let d be the set of data points with the condition such that a = v.
11. If there are no data points in d
12. Create a leaf with the majority class.
13. Else repeat step 1 with d data points.
**Naïve Bayes**

These are statistical classifiers. They can predict class membership probabilities, such as the probability that a given sample belongs to a particular class. Because it is probabilistic and involves relatively simple calculation of probabilities, Bayesian classifiers require less training time compared to sophisticated algorithms like neural networks. But studies comparing Naïve Bayes classifier with decision tree and neural networks have found the performance or Naïve Bayes to be comparable with the two.

Naïve Bayes classifiers assume that the effect of an attribute value on a given class is independent of the values of other attributes. The assumption is called class conditional independence. The assumption helps simplify computations and thus makes the algorithm fast.

**Steps in Naïve Bayes classification**

1. We have an unknown data sample X (without class information) with n attributes and values x1, x2, ..., xn for its attributes. Let us call the attributes a1, a2, ..., an.
2. There are m classes c1, c2, ..., cm in the entire data. Given X, we predict X to belong to class ck such that P(ck/X) is maximum for all k from 1 to m.
4. P(X) prior probability of X, does not depend on class information and thus, we only need to maximize P(ck) and P(X/ck).
5. Given class probabilities, calculate P(ck) = sk/s (where sk is the number of data points belonging to class k and s is the total number of data points).
6. To maximize P(X/ck), we have to calculate P(X/ci), for all i = 1 ... m and pick the maximum.
7. By class conditional independence we calculate each P(X/ci) to be the product of P(xj/ci), where j = 1 ... n.
Neural Networks

Neural Net approach is based on a neural network, which is a set of connected input/output units (perceptrons) where each connection has a weight associated with it. During the learning phase the network learns by adjusting the weights so as to be able to predict correct target values. Backpropagation is one of the most popular neural net learning methods. The backpropagation algorithm performs learning on multi layer feed forward neural network. The input to the network is a data point and the number of nodes in the input layer is the same as the number of attributes in the data point (i.e., if there are i conditional attributes there should be i nodes.). The inputs represent the values of attributes for the data point. The inputs are fed simultaneously into a layer of units making up the input layer. The weighted outputs of these units are in turn fed simultaneously to a second layer known as hidden layer. The hidden layers weighted output can be fed to a next hidden layer and so on. But in practice usually only one hidden layer is used. Layers are connected by weighted edges. The weights Wij, are adjusted when the prediction of the network does not match the actual class value. (labeled data is required, meaning data whose class value is known should be used for training). The modification of weights is from the output to the previous hidden layer and

IV. EFFORT PREDICTION IN SOFTWARE PROJECTS

Many commentators have suggested the use of more than one technique in order to support effort prediction, but to date there has been little or no empirical investigation to support this recommendation. Our analysis of effort data from a medical records information system reveals that there is little, or even negative, covariance between the accuracy of our three chosen prediction techniques, namely, expert judgment, least squares regression and case-based reasoning. This indicates that when one technique predicts poorly, one or both of the others tends
to perform significantly better. This is a particularly striking result given the relative homogeneity of our data set. Consequently, searching for the single "best" technique, at least in this case, leads to a sub-optimal prediction strategy. The challenge then becomes one of identifying a means of determining a priori which prediction technique to use. Unfortunately, despite using a range of techniques including rule induction, we were unable to identify any simple mechanism for doing so.

**Data collection procedure**

The next step is to collect metrics data from the systems. In the target environment, all software systems developed were accompanied by final documentation. This documentation contained an ERD and FHD for the final system and effort data recorded by each developer in the group. All forms, reports and graphs in the final system and the database were also stored in a repository provided by the development tool suite. The required metrics data were collected from those sources. During this process, two systems were eliminated due to the incomplete effort data. This resulted in the remaining 17 systems being used in this study, all of which have the same number of developers. The productivity metric was calculated by using the average mark of the developers in each team from a practical development test undertaken in the target environment.

**V. DATA ANALYSIS**

**Descriptive Statistics**

The next step is to analyze the collected metrics data. The effort data are measured in hours. The differences observed between the medians and means, and the values of the skewness statistic show that the data are skewed. Therefore, in the following exploratory analysis, non-parametric techniques, which do not require the normality assumption, are used.
Correlation Analysis

In order to examine the existence of the potential linear relationships between the specification-based software size metrics and development effort, and the degree of linear association between the specification-based software size metrics themselves, correlation analysis was performed. Spearman’s rank correlation coefficient was used in this analysis. When correlated metrics are included in a regression model, multicollinearity can cause some difficulty for users in interpreting some partial correlation coefficients and increases the standard errors of the predicted values. Thus, when constructing a multivariate regression model, it is recommended to include software size metrics which are not highly correlated with each other. Principal component analysis (PCA) can be used to construct independent variables using linear transformations of the original input variables. However, PCA is not used in this study as neither the difficulty in interpretation nor the problem of the errors is identified in the models.

VI. CONCLUSIONS

Since IT Organizations / Management use cost / effort estimates to approve or reject a project proposal or to manage the maintenance process more effectively, it is very important that scientific / empirical models be used for effort / defect predictions for an effective estimation of Projects. Furthermore, accurate cost estimates would allow organizations to make more realistic bids on external contracts. Unfortunately, effort estimation is one of the most relevant problems of the software maintenance process. Predicting software maintenance effort is complicated by many typical aspects of software and software systems that affect maintenance activities. So it is evident that these scientific models are indispensable in Software estimations.
VII. REFERENCES


