© UNIVERSAL RESEARCH REPORTS | REFEREED | PEER REVIEWED
ISSN : 2348 - 5612 | Volume : 04 , Issue : 10 | October – December 2017

Study Of Approaches To Test Effort Estimation , Metrics Used For Software Project Size Estimation And Shortcomings Of Function Point (FP) Metric

[1]**Nidhi Kandhil** nidhikandhil@gmail.com
[2]**Vikas Chahar,** Head, Department of Computer Science Vaish (P.G.) College, Rohtak

Abstract— Software testing effort estimation has always been an on-going challenge to software engineers, as testing is one of the critical activities of SDLC. Accurate effort estimation is the state of art of software engineering, as it is the preliminary phase between the client and the business enterprise. The credibility of the client to the business enterprise increases with the accurate estimation. The earlier test estimation is done, the more benefits will be achieved in the testing life cycle. This paper proposes an approach for estimating the size and efforts required in the testing projects using test case point. The proposed model outlines all major factors that affect testing projects. Covering all major factors helps to do a fair estimation using the proposed approach. Many times, software test teams are responsible to handle different test activities such as estimation of test effort, bug identification, test case design, test tool selection, test team selection, etc. Hence, test managers need to plan the schedule accurately and efficiently to utilize the testing resources in order to meet deadlines. An accurate and efficient test effort estimation method could help test managers in completing projects successfully. Inaccurate effort estimates may lead to poor quality, customer dissatisfaction, and developers' frustration. Project uncertainty, use of estimation development processes, use of estimation management processes and the estimator's experience, are a few factors that can affect effort estimation errors.

Key Words : Software testing effort estimation, LOC, FP etc.

Related Work

In recent years, several researchers have used Bayesian network (BN) to model uncertainties in software projects, e.g. BNs have been used successfully to support the managerial decision-making , allowing the project manager to trade-off the resources used against the output in terms of functionality and quality . What follows next is a summary of relevant work in this field.

In 2013 Shaik Nafeez Umar studied "SOFTWARE TESTING DEFECT PREDICTION MODEL-A PRACTICAL APPROACH " and found that Software defects prediction aims to reduce software testing efforts by guiding the testers through the defect classification of software systems. Defect predictors are widely used in many organizations to predict software defects in order to save time, improve quality, testing and for better planning of the resources to meet the timelines. The application of statistical software testing defect prediction model in a real life setting is extremely difficult because it requires more number of data variables and metrics and also historical defect data to predict the next releases or new similar type of projects. This paper explains our statistical model, how it will accurately predict the defects for upcoming software releases or projects. We have used 20 past release data points of software project, 5 parameters and build a model by applying descriptive statistics, correlation and multiple linear

regression models with 95% confidence intervals (CI). In this appropriate multiple linear regression model the R-square value was 0.91 and its Standard Error is 5.90%. The Software testing defect prediction model is now being used to predict defects at various testing projects and operational releases. We have found 90.76% precision between actual and predicted defects.

In 2015 Mridul Bhardwaj , Ajay Rana studied " Estimation of Testing and Rework Efforts for Software Development Projects" and found that In the planning of a software development project, a major challenge faced by project managers is to predict the rework effort. (Rework effort is the effort required to fix the software defects identified during system testing). The project manager's objective is to deliver the software within the time, cost and quality requirements given by the client. To ensure the quality of the software, many testing cycles will beconducted before it is finally delivered to the client for acceptance. Each testing cycle is a costly affair as it involves running all possible test scenarios in all possible environments, followed by defect fixing and re-verification of defects. On average, two to three testing cyclesare conducted but this depends on the number of defects identified during testing.

In 2002 E. Mendes et. al. studied A comparison of development effort estimation techniques for Web hypermedia applications" and observed that Several studies have compared the prediction accuracy of different types of techniques with emphasis placed on linear and stepwise regressions, and case-based reasoning (CBR). We believe the use of only one type of CBR technique may bias the results, as there are others that can also be used for effort prediction. This paper has two objectives. The first is to compare the prediction accuracy of three CBR techniques to estimate the effort to develop Web hypermedia applications. The second objective is to compare the prediction accuracy of the best CBR technique, according to our findings, against three commonly used prediction models, namely multiple linear regression, stepwise regression and regression trees. One dataset was used in the estimation process and the results showed that different measures of prediction accuracy gave different results. MMRE and MdMRE showed better prediction accuracy for multiple regression models whereas box plots showed better accuracy for CBR.

In 1997 G.R.Finnies studied " A comparison of software effort estimation techniques: Using function points with neural networks, case-based reasoning and regression models" and observed that Estimating software development effort remains a complex problem attracting considerable research attention. Improving the estimation techniques available to project managers would facilitate more effective control of time and budgets in software development. This paper reviews a research study comparing three estimation techniques using function points as an estimate of system size. The models considered are based on regression analysis, artificial neural networks and case-based reasoning. Although regression models performed poorly on the data set of 299 projects, both artificial neural networks and case-based reasoning appeared to have value for software development effort estimation models. Case-based reasoning in particular is appealing because of its similarity to expert judgment approaches and for its potential as an expert assistant in support of human judgment.

In , Rees et al. used Bayesian (graphical) models to model the uncertainties involved in software testing, quality process and provide support to test managers to use the model. Fenton and Neil , and Hearty et al. , have all shown that BNs have many benefits over classical or regression-based models. The Bayesian

network approach does not rely on a single point value; instead of predicting a single value of the variable a BN provides a complete probability distribution.

In 2006, Wang et al. presented a project level estimation model framework using a Bayesian belief network (BBN) . Their BBN used four sub-models: component estimation; test effectiveness estimation; residual defect estimation; and test estimation. By using this framework, estimation of quality, effort, schedule and scope could be determined at both project level and in specific phases; hence, providing support for managerial decision-making. However, the problem with Wang's et al. framework was that it had not been validated in any real project and, further, was not tailored to any particular development methodology. Moreover, no statistical and prediction accuracy measures were performed.

In 2009, Hearty et al. , presented a Bayesian network causal model for the extreme programming (XP) methodology. They showed how, without using any additional metrics collection programs, their model could learn from project data in order to make quantitative effort predictions and risk assessments. They validated their model in an industry project. However, the model was validated with one instance of project data and only two XP practices were introduced into the model. They focused on project level predictions rather than specific testing processes.

Test Estimation

Test Estimation is the estimation of the testing size, testing effort, testing cost and testing schedule for a specified software testing project in a specified environment using defined methods, tools and techniques.

1. Estimation – defined in the earlier chapters
2. Testing Size – the amount (quantity) of testing that needs to be carried out.
3. Testing Effort – the amount of effort in either person days or person hours necessary for conducting the tests
4. Testing Cost – the expenses necessary for testing, including the expense towards human effort
5. Testing Schedule – the duration in calendar days or months that is necessary for conducting the tests

Approaches to Test Effort estimation : The following approaches are available for carrying out Test Effort Estimation

1. Delphi Technique
2. Analogy Based estimation
3. Software Size Based Estimation
4. Test Case Enumeration Based Estimation
5. Task (Activity) based Estimation
6. Testing Size Based Estimation

Metrics for software project size estimation

Accurate estimation of the problem size is fundamental to satisfactory estimation of effort, time duration and cost of a software project. In order to be able to accurately estimate the project size, some important metrics should be defined in terms of which the project size can be expressed. The size of a problem is obviously not the number of bytes that the source code occupies. It is neither the byte size of the

executable code. The project size is a measure of the problem complexity in terms of the effort and time required to develop the product.

Currently two metrics are popularly being used widely to estimate size: lines of code (LOC) and function point (FP). The usage of each of these metrics in project size estimation has its own advantages and disadvantages.

- lines of code (LOC)
- Function point (FP)

Lines of Code (LOC)

LOC is the simplest among all metrics available to estimate project size. This metric is very popular because it is the simplest to use. Using this metric, the project size is estimated by counting the number of source instructions in the developed program. Obviously, while counting the number of source instructions, lines used for commenting the code and the header lines should be ignored.

Determining the LOC count at the end of a project is a very simple job. However, accurate estimation of the LOC count at the beginning of a project is very difficult. In order to estimate the LOC count at the beginning of a project, project managers usually divide the problem into modules, and each module into submodules and so on, until the sizes of the different leaf-level modules can be approximately predicted. To be able to do this, past experience in developing similar products is helpful. By using the estimation of the lowest level modules, project managers arrive at the total size estimation.

Function point (FP)

Function point metric was proposed by Albrecht [1983]. This metric overcomes many of the shortcomings of the LOC metric. Since its inception in late 1970s, function point metric has been slowly gaining popularity. One of the important advantages of using the function point metric is that it can be used to easily estimate the size of a software product directly from the problem specification. This is in contrast to the LOC metric, where the size can be accurately determined only after the product has fully been developed.

The conceptual idea behind the function point metric is that the size of a software product is directly dependent on the number of different functions or features it supports. A software product supporting many features would certainly be of larger size than a product with less number of features. Each function when invoked reads some input data and transforms it to the corresponding output data. For example, the issue book feature of a Library Automation Software takes the name of the book as input and displays its location and the number of copies available. Thus, a computation of the number of input and the output data values to a system gives some indication of the number of functions supported by the system. Albrecht postulated that in addition to the number of basic functions that a software performs, the size is also dependent on the number of files and the number of interfaces.

Besides using the number of input and output data values, function point metric computes the size of a software product (in units of functions points or FPs) using three other characteristics of the product as

shown in the following expression. The size of a product in function points (FP) can be expressed as the weighted sum of these five problem characteristics. The weights associated with the five characteristics were proposed empirically and validated by the observations over many projects. Function point is computed in two steps. The first step is to compute the ***unadjusted function point (UFP).***

UFP = (Number of inputs)*4 + (Number of outputs)*5 + (Number of inquiries)*4 + (Number of files)*10 + (Number of interfaces)*10

Shortcomings of function point (FP) metric : *LOC as a measure of problem size has several shortcoming*

- LOC gives a numerical value of problem size that can vary widely with individual coding style – different programmers lay out their code in different ways. For example, one programmer might write several source instructions on a single line whereas another might split a single instruction across several lines. Of course, this problem can be easily overcome by counting the language tokens in the program rather than the lines of code. However, a more intricate problem arises because the length of a program depends on the choice of instructions used in writing the program. Therefore, even for the same problem, different programmers might come up with programs having different LOC counts. This situation does not improve even if language tokens are counted instead of lines of code.
- A good problem size measure should consider the overall complexity of the problem and the effort needed to solve it. That is, it should consider the local effort needed to specify, design, code, test, etc. and not just the coding effort. LOC, however, focuses on the coding activity alone; it merely computes the number of source lines in the final program. We have already seen that coding is only a small part of the overall software development activities. It is also wrong to argue that the overall product development effort is proportional to the effort required in writing the program code. This is because even though the design might be very complex, the code might be straightforward and vice versa. In such cases, code size is a grossly improper indicator of the problem size.
- LOC measure correlates poorly with the quality and efficiency of the code. Larger code size does not necessarily imply better quality or higher efficiency. Some programmers produce lengthy and complicated code as they do not make effective use of the available instruction set. In fact, it is very likely that a poor and sloppily written piece of code might have larger number of source instructions than a piece that is neat and efficient.
- LOC metric penalizes use of higher-level programming languages, code reuse, etc. The paradox is that if a programmer consciously uses several library routines, then the LOC count will be lower. This would show up as smaller program size. Thus, if managers use the LOC count as a measure of the effort put in the different engineers (that is, productivity), they would be discouraging code reuse by engineers.
- LOC metric measures the lexical complexity of a program and does not address the more important but subtle issues of logical or structural complexities. Between two programs with equal LOC count, a program having complex logic would require much more effort to develop than a program with very simple logic. To realize why this is so, consider the effort required to

develop a program having multiple nested loop and decision constructs with another program having only sequential control flow.

- It is very difficult to accurately estimate LOC in the final product from the problem specification. The LOC count can be accurately computed only after the code has been fully developed. Therefore, the LOC metric is little use to the project managers during project planning, since project planning is carried out even before any development activity has started. This possibly is the biggest shortcoming of the LOC metric from the project manager's perspective.

References :

1. Ali Idri, Azeddine Zahi, "Software cost estimation by classical and Fuzzy Analogy for Web Hypermedia Applications: A replicated study", Computational Intelligence and Data Mining (CIDM) 2013 IEEE Symposium on, pp. 207-213, 2013.
2. Emilia Mendes, Nile Mosley, "Bayesian Network Models for Web Effort Prediction: A Comparative Study", Software Engineering IEEE Transactions on, vol. 34, pp. 723-737, 2008, ISSN 0098-5589.
3. E. Mendes, N. Mosley, S. Counsell, "A replicated assessment of the use of adaptation rules to improve Web cost estimation", Empirical Software Engineering 2003. ISESE 2003. Proceedings. 2003 International Symposium on, pp. 100-109, 2003.
4. A comparison of software effort estimation techniques: Using function points with neural networks, case-based reasoning and regression models, Journal of Systems and Software, Volume 39, Issue 3, 1997, Pages 281-289, ISSN 0164-1212, https://doi.org/10.1016/S0164-1212(97)00055-1.
5. N. E. Fenton, P. Krause, and M. Neil, "Software measurement: Uncertainty and causal modeling," IEEE Software, vol. 19, no. 4, pp. 116–122, 2002.
6. D. Settas, S. Bibi, P. Sfetsos, I. Stamelos, and V. Gerogiannis, "Using Bayesian belief networks to model software project management antipatterns," in SERA '06: Proceedings of the Fourth International Conference on Software Engineering Research, Management and Applications. Washington, DC, USA: IEEE Computer Society, 2006, pp. 117–124.
7. N. E. Fenton, W. Marsh, M. Neil, P. Cates, S. Forey, and M. Tailor, "Making resource decisions for software projects," in ICSE '04: Proceedings of the 26th International Conference on Software Engineering. Washington, DC, USA: IEEE Computer Society, 2004, pp. 397–406.
8. N. E. Fenton and M. Neil, "A critique of software defect prediction models," IEEE Transactions on Software Engineering, vol. 25, no. 5, pp. 675–689, 1999.
9. P. Hearty, N. E. Fenton, D. Marquez, and M. Neil, "Predicting project velocity in XP using a learning dynamic Bayesian network model," IEEE Transactions on Software Engineering, vol. 35, no. 1, pp. 124–137, 2009.
10. F. V. Jensen and T. D. Nielsen, Bayesian networks and decision graphs. Springer Publishing Company, Inc., 2007.
11. N. E. Fenton and S. L. Pfleeger, Software metrics: A rigorous and practical approach. Boston, MA, USA: International Thomson Computer Press, 1996.