International Journal of Engineering and Advanced Technology (IJEAT)
ISSN: 2249-8958 (Online), Volume-8 Issue-6S3, September 2019

# Machine Learning Algorithm Application in Software Quality Improvement using Metrics

**Dharmapuri. Siri**

***Abstract:*** ***Machine learning purely concerned on the concept with building the program that improves the tasks performance through experience. Machine learning algorithms have proven to be of great practical value in a variety of application domains. the field of software engineering turns out to be a fertile ground where many software development tasks could be formulated as learning problems, analyzing design and testing plays the major role and approached in terms of learning algorithms We discuss several metrics in each of five types of software quality metrics: product quality, in-process quality, testing quality, maintenance equality, and customer satisfaction quality.***

## I. INTRODUCTION

We investigate the problem of making machine learning (ML) applications dependable, focusing on software testing. Software engineering processes and tools do not neatly apply: in particular, it is challenging to detect subtle errors, faults, defects or anomalies [henceforth "bugs"] in the ML applications of interest because there is no reliable "test oracle" to indicate what the correct output should be for arbitrary input. The general class of software systems with no reliable test oracle available is sometimes known as "non-testable programs". These ML applications fall into a category of software that Davis and Weyuker describe as "*Programs which were written in order to determine the answer in the first place. There would be no need to write such programs, if the correct answer were known"*. Formal proofs of an ML algorithm's optimal quality do not guarantee that an application implements or uses the algorithm correctly, and thus software testing is needed. Our testing, then, does not seek to determine whether an ML algorithm learns well, but rather to ensure that an application using the algorithm correctly implements the specification and fulfills the users' expectations. In this paper, we describe our approach to testing ML applications, in particular those that implement ranking algorithms (a requirement of the real-world problem domain). Of course, in any software testing, it is possible only to show the presence of bugs but not their absence. Usually when input or output equivalence classes are applied to developing test cases, however, the expected output for a given inputs known in advance. Our research speaks about how one can device test cases, that expose bugs, and also how we indeed know "Do a test is revealing a bug , Actually?", given that we do not know what the output should be in the general case.

**Existing procedure:**
In all the existing system they had used machine learning algorithm to improve the software quality but there is not measurement for this software engineering process and each and every task in this process is not measured.
Here are some conference references:

## II. THE RESEARCH ON SOFTWARE METRICS AND SOFTWARE COMPLEXITY METRICS CONFERENCE

**I. 25-27 DEC. 2009CHONGQING, CHINA.**

**II. IN THE ABOVE PAPER THE "SOFTWARE METRICS DEFINITIONS WERE GIVEN AND SOFTWARE METRICS WERE OVERVIEWED. SOFTWARE COMPLEXITY MEASURING IS THE IMPORTANT CONSTITUENT OF SOFTWARE METRICS AND IT IS CONCERNING THE COST OF SOFTWARE DEVELOPMENT AND MAINTENANCE".**

[2] A Survey on Software Quality Metrics Thorbj¨orn Andersson ˚Abo Akademi University Department of Computer Science SF-20520 Turku, Finland December 4, 1990 By this survey has been written with specially the quality factors maintainability and reliability and the quality criterion complexity in mind. It is geared toward my future research interest of ensuring quality of software by automated data collection of the software quality metrics

**III. [3]METRICS FOR MEASURING THE EFFECTIVENESS OF SOFTWARE-TESTING TOOLS**

J.B. Michael ; B.J. Bossuyt ; B.B. Snyder
Conference**:** 12-15 Nov. 2002 Annapolis, MD, USA
From this paper, they proposed a suite of objective metrics for measuring tool characteristics, as an aid in systematically evaluating and selecting automated testing tools.

**IV. [4] SOFTWARE TESTING AND METRICS FOR CONCURRENT COMPUTATION AUTHOR(S)T.K. SHIH ; CHI-MING CHUNG ; ET AL SEOUL, SOUTH KOREA, SOUTH KOREA 4-7 DEC. 1996**

**V. FROM THIS PAPER, THEY PROPOSED FOUR TESTING CRITERIA TO TEST A CONCURRENT PROGRAM. A PROGRAMMER CAN CHOOSE AN APPROPRIATE TESTING STRATEGY DEPENDING ON THE PROPERTIES OF THE CONCURRENT PROGRAMS. ASSOCIATED WITH THE STRATEGIES, FOUR EQUATIONS ARE PROVIDED TO MEASURE THE COMPLEXITY OF CONCURRENT PROGRAMS.**

**Manuscript published on 30 September 2019.**
***** Correspondence Author (s)
**Dharmapuri.siri,** Department of information technology, Malla reddy engineering college for women Hyderabad, india Scholar of jjtu Emails: dharmapuri.siri@gmail.com

**Proposed procedure:**

In this paper I am purely dealing with software engineering process using machine learning technique and each and every task and its measurement.

## III. MACHINE LEARNING ALGORITHMS

Machine learning deals with the issue of how to build computer programs that improve their performance at some task through experience. Machine learning algorithms have been utilized in:(1) data mining problems where large databases may contain valuable implicit regularities that can be discovered automatically.(2) poorly understood domains where humans might not have the knowledge needed to develop effective algorithms. And (3) domains where programs must dynamically adapt to changing conditions. Learning a target function from training data involves many issues (function representation, how and when to generate the function, with what given input, how to evaluate the performance of generated function, and so forth).

**Table. SE tasks and applicable ML methods**.

| SE tasks | Applicable type(s) of learning |
|---|---|
| Requirement engineering | AL, BBN, LL, DT, ILP |
| Rapid prototyping | GP |
| Component reuse | IBL (CBR[4]) |
| Cost/effort prediction | IBL (CBR), DT, BBN, ANN |
| Defect prediction | BBN |
| Test oracle generation | AL (EBL[5]) |
| Test data adequacy | CL |
| Validation | AL |
| Reverse engineering | CL |

## IV. SOFTWARE QUALITY METRICS

### *1) Productive Quality Metrics*

Lewis and Henry (1990) "divide the software complexity metrics into three categories are: Code metrics, structure metrics, and hybrid metrics. Code metrics examine the internal complexity of a procedure. Example of code metrics are LOC, Heal stead's software science, and McCabe's Cyclometric complexity. Structure metrics examine the relationship between a section code and the rest of the system. Example of Structure metrics are Information flow metrics. 'The hybrid metrics are combined the internal code metrics with of the communication connection s between the code and the rest of the system'.

### *2) Example 1: Lines of Code Defect Rate*

Because the LOC count is based on source instructions, then there are two size metrics are shipped source instruction (SSI) and new and changed source instructions (CSI). The relationship between the SSI and CSI count can be expressed with the following formula:

SSI (current release) = SSI (previous release) +CSI (new and changed source instructions for release) – Deleted code (usually very small) –Changed code (to avoid double count in both SSI and CSI)

The several post-release defect rate metrics per thousand SSI (KSSI) or per thousand CSI (KCSI) are:

(1) Total defects pear KSSI (a measure of code quality of the total product)

(2) Field defects per KSSI ( a measure of defect rate in the field)

(3) Release-origin defects (field and internal) per KCSI ( a measure of development quality)

(4) Release-origin field defects per KCSI ( a number of development quality per defects found by customers)

Consider the following hypothetical example:

Initial release of product X

KCSI = KSSI = 50KLOC

Defects / KCSI = 2.0

Total number of defects =2.0 x 50 = 100 Second release of product X

KCSI = 20

KSSI = 50 + 20 (new and changed lines of code) – 4 (assuming 20% are changed) Line of codes) = 66

Defects / KCSI = 1.8 (assuming 10% improvement over the first release) Total number of defects =1.8 x 20 = 36

**Process Quality Metrics**

**Defect Density during Testing:**

Defect rate during formal testing is usually positively correlated with the defect rate in the field. Higher defect rates found during testing in an indicator that the software has experienced higher error injection during testing effort – for example, additional testing or a new testing approach that was demand more effective in detecting defects. Some metrics for defect density during testing are:

◦ Error discovery rate: number of total defects found / number of test procedures execution.

◦ Defect acceptance: (Number of valid defects / total number of defects) * 100

◦ Test case defect density: (Number of failed tests / Number of executed test cases)*100

**Defect Arrival / removal During Testing:**

The objective is always to look for defect arrivals that stabilize at a very low level, or times between failures that are far apart, before ending effort and releasing the software to the field. Some metrics for defect arrival during testing are:

◦ Bad Fix defect: defect whose resolution give rise to new defects are bad fix defect. Bad Fix defect = (Number of Bad Fix defects / Total number of valid defects)*100



Published By:
Blue Eyes Intelligence Engineering
and Sciences Publication (BEIESP)

◦ Defect removal effectiveness (DRE): (Defects removed during a development phase / Defects latent in the product)* 100. The denominator of the metric can only be approximated by defects removed during the phase + defects found later.

**Metric-based Estimation Models:**

Most of the models presented in this subsection are estimators of the effort needed to produce a software product. Probably the best known estimation model is Boehm's COCOMO model (Boehm, 1981). "The first one is a basic model which is a single-value model that computes software development effort and cost as a function of program size expressed as estimated lines of code (LOC). The second COCOMO model computes software development effort as a function of program size and a set of "coat drives" that include subjective assessment of product, hardware, personal, and project attributes. The third COCOMO model is an advanced model that incorporates all characteristics of the intermediate version with the assessment of the cost driver's impact on each step of the software engineering process".
Given in Table

The basic COCOMO equations are: $E \square a_i (KLOC)^{b_i}$ ,

Where E is theeffort applied in person-month.

$D = c_i E^{d_i}$

D is the development time ichrono logical month

The coefficients $a_i$ and $c_i$ and the exponents $b_i$ and $d_i$ are given in table

***3) Table. Basic COCOMO***

| Software project | $a_i$ | $b_i$ | $c_i$ | $d_i$ |
|---|---|---|---|---|
| Organic | 2.4 | 1.05 | 2.5 | 0.36 |
| Semi-detached | 3.0 | 1.12 | 2.5 | 0.35 |
| Embedded | 3.6 | 1.20 | 2.5 | 0.32 |

The Putnam estimation model (Putnam, 1978) assumes a specific distribution of effort over the software development project. The distribution of effort can be described by the Royleigh- Norden curve. The equation is:

$L = c_k K^{1/3} t_d 4/3$

Where $c_k$ is the state of technology constant,

$k$ is the effort expended (in person-years) over the whole life cycle.

$t_d$ is the development time in year.

Walston and Fellix (1977) give a productivity estimator of a similar form at their document metric. The programming productivity is defined as the ratio of the delivered source lines of code to the total effort in person-months required to produce the delivered product.

$E = 5.2L^{0.91}$ E

Where E is total effort in person-month

L is the number of 1000lines of code

*4) Software Maintenance Metrics*

During the maintenance phase, the following metrics are very important:

- Fix backlog and backlog management index
- Fix response time and fix responsiveness
- Percent delinquent fixes
- Fix quality

Fix backlog is a workload statement for software maintenance. To manage the backlog of open, unresolved, problems is the backlog management index (BMI). If BMI is large then 100, it means the backlog is reduced. If BMI is less than 100, then the backlog increased.

$$BMI = \frac{Numberof\ problrmsclosed\ duringthe\ month}{Numberof\ problemarrivals\ during\ the\ month} \times 100\%$$

The fix response time metric is usually calculated as follows for all problems as well as by severity level: Mean time of all problems from open to close. A more sensitive metrics is the percentage of delinquent fix. For each fix, if the turnaround time greatly exceeds the require response time, then it is classified as delinquent:

$$\text{delinquent Percent} = \frac{numberoffixes\ that\ exceeded\ the\ \text{fixes}\ responsetimecriticiaby\ severitylevel}{Numberoffixes\ deliveredin\ a\ specifiedtime}$$

This metrics is not a metric for real-time delinquent management because it is for closed problem only.
Fix quality or the number of defective fixes is another important quality metric for the maintenance phase.

*5) Software Testing Metrics*

The software metrics that the quality assurance (QA) team procedures are connected with the test activities that are part of test phase and so are formally known as software testing metrics (Kaur *et al.*, 2007).

*6) Example:*

In this production, average response time is greater than expected, requirement met during perform test = 4, requirement not met after signoff of perform test = 1. Consider, average response time is important requirement which has not met, then tester can open defect with severity as critical. "Performance severity index" = (4* 1)/1 = 4 (critical). "Performance test efficiency" = 4/ (4+1)* 100 = 80%.



Published By:
Blue Eyes Intelligence Engineering
and Sciences Publication (BEIESP)

*7) Customer Problems Metric and Customer Satisfaction Metrics*

From the customer's perspective," it is bad enough to encounter functional defects when running a business on the software". The problems metric is usually expressed in terms of problem per user month (PUM). PUM is usually calculated for each month after the software is released to market, and also for monthly averages by user. The customer problems metric can be regarded as an intermediate measurement between defects measure and customer satisfaction. To reduce customer problems, one has to reduce the functional defects in the products, and improve other factors (usability, documentation, problem rediscovery, *etc.*). To improve customer satisfaction, one has to reduce defects and overall problems. Several metrics with slight variations can be Constructed and used, depending on the purpose of analysis.

**For example:**

- Percent of completely satisfied customers.
- Percent of satisfied customers (satisfied and completely satisfied)
- Percent of dissatisfied customers(dissatisfied and completely dissatisfied)
- Percent of non (neutral, dissatisfied, and completely dissatisfied).

## V. CONCLUSIONS

In this paper we discuss the software quality improvement using machine learning algorithm with software tasks .Software measurement and metrics is useful for us a lot of evaluating software process as well as the software product. The set of measures identified in this paper provide the organization with better insight into the validation activity, improving the software process towards the goal of the having a management process.

## REFERENCES


1. L. J. Arthur, "Measuring programmer productivity and software quality", John Wiley & Son, NY, (1985).
2. J. H. Baumert and M. S. McWihinnet, "Software measurement and the capability maturity model", Software Engineering Institute Technical Report, cMMI/SEI-92-TR, ESC-TR-92-0, (1992).
3. B. W. Boehm, "Software Engineering Economics", Englewood Cliffs, NJ, Prentice Hall, (1981).
4. M. Bundschuh and A. Fabry, "Auswandschatzung von IT-project",
5. M. Dao, M. Huchard, T. Libourel and H. Leblance, "A new approach to factorization-introducing metrics", Proceeding of the IEEE Symposium on Software Metrics METRICS, (2002) June 4-7, pp. 227-236.
6. R. Dumake, M. Lother and C. Wille, "Situation and treads in Software Measurement-A statistical Analysis of the SML Metrics Biolography, Dumke / Abran: Software Measurement and Estimation, Shaker Publisher, (2002), pp. 298-514.
7. R. Dumake, M. Lother and C. Wille, "Situation and treads in Software Measurement-A statistical Analysis of the SML Metrics Biolography, Dumke / Abran: Software Measurement and Estimation, Shaker Publisher, (2002), pp. 298-514.
8. S. U. Farooq and A. M. K. Quadri, "Software measurements and metrics: Role in effective software testing", International Journal of Engineering Science and Technology, vol. 3, no. 1, (2011), pp. 671-680.
9. Conference, 1981, pp. 254-257.
10. P. Long and R. Servedio, "Martingale Boosting",Eighteenth Annual Conference on Computational Learning Theory (COLT), Bertinoro, Italy, 2005, pp. 79-94.
11. T. Joachims, Making large-Scale SVM Learning Practical. Advances in Kernel Methods - Support Vector
12. Learning, B. Schölkopf and C. Burges and A. Smola (ed.),MIT-Press, 1999