Metrics and Prediction Methods for Software Quality: A Review

Alli Said Rashid
Faculty of Electrical and Computer Engineering
University of New Brunswick
Athens, Greece
alli.rashid@unb.ca

Mohammadali Rahnama
Faculty of Computer Science
University of New Brunswick
Fredericton, Canada
m.rahnama@unb.ca

***Abstract*—This paper provides a historical overview of software metrics, beginning with the classical paradigm and progressing through the object-oriented, component-based, and aspect-oriented levels of abstraction. Defects in software may arise at any stage of development, from initial planning to final testing. There are multiple points in the software creation process where predicting the final product's quality is essential. One of the most crucial indicators of software quality is its failure probability. When it comes to making decisions, quality measurements for software are crucial. They can be used as a means of communicating information about a program or its development and are represented by numerical numbers. The code's ability to pass tests or its complexity are two areas that might be highlighted by certain metrics. Developer productivity can also be gauged by keeping tabs on how much time is spent on various activities like planning, coding, and bug fixing. Because high productivity is necessary for a product to be delivered at a sustainable pace, measuring it can help decide how large a development team should be. Furthermore, we described two distinct software quality prediction algorithms, namely, decision trees in machine learning and feature selection. Bugs in software are a common occurrence, therefore high-assurance software systems are built to catch them early on in the development process and fix them before they are released to the public. Ultimately, the field of software engineering has been the topic of many studies aimed at enhancing the quality of the final software output.**

Keywords—software quality metrics, quality prediction, machine learning, decision trees, feature selection

I. Introduction

The quality of a software product can be evaluated with the help of a set of metrics known as "software quality metrics." Quality assurance metrics like test coverage and agile metrics like velocity are two such examples. Metrics provide valuable insight into the visible aspects of software quality by measuring characteristics such as size, complexity, and reliability. In this paper, we trace the development of software metrics from the classical to the object-oriented to the component-based to the aspect-oriented paradigms. Furthermore, the article explains how the adoption of these new paradigms has boosted software quality, system performance, and overall productivity [1].

Software defects can occur in a variety of places, including requirements analysis, specification, and implementation. Predicting software quality is a necessary task at several stages of the development process. Two potential applications are project-based quality assurance practices planning and benchmarking. The probability of failure is one of the most important metrics for evaluating software quality [1].

Software quality metrics are critical when it comes to management. They are numerical values that can convey information about a program or its progression. Some metrics place emphasis on the complexity of the code or its ability to pass tests when evaluating its quality. The efficiency of developers can also be measured by tracking how much time they devote to various tasks such as planning, coding, and bug fixing. Project managers can use metrics to identify inefficiencies and make necessary adjustments [2].

A software system's software quality metrics can be identified, implemented, analyzed, and validated with the help of the IEEE Standard for a Software Quality Metrics Methodology. It covers the entire software development process and consists of five stages [3]:

First, determine what needs to be done to ensure high-quality software production. This includes selecting, ranking, and quantifying a set of quality factors to be used in the system's development or modification.

Second, find metrics for software quality by utilizing the software quality metrics framework to determine the best metrics to use.

Third, put the software quality metrics into action by acquiring or developing the necessary tools, collecting the necessary data, and applying the metrics to each stage of the software development life cycle.

Fourth, analyze the results of software quality metrics and report on the findings to aid in development management and product evaluation.

Finally, the software quality metrics are validated by comparing the results of the predictive metrics to the results of the direct metrics to see if the predictive metrics successfully measure the factors they are supposed to measure.

The remaining sections of this report are organized as follows: Section II contains a comprehensive list of software quality metrics, variants on those metrics, and specific details on how each metric listed is measured. Section III includes a review of software quality prediction as well as a description of two techniques in predicting software quality. Section IV concludes the paper by summarizing the findings and interpretations.

II. Metrics for Evaluating Software Quality

A. Agile Metrics: Lead Time, Cycle Time, Velocity

The introduction of Agile software development has certainly provided companies with an additional method for efficiently developing products. Agile focuses on an iterative approach to development through short sprints and systematic collaboration. Through regular collaboration and communication, teams are better prepared to handle changes in product requirements, hence the term Agile. The goal at the end of every sprint, a period in which a team works to

complete a set of tasks, is to provide a functioning version of the application that contains preferably no defects. The following agile metrics should be considered to ensure software quality in projects: lead time, cycle time, and velocity .

"Lead time is the period starting when the task or project is requested and ending with the completion of the assignment" [4]. Additionally, it includes the amount of time it takes to develop a business requirement and correct code defects. This metric is significant in providing an accurate time span of completion for every task in a sprint.

"Cycle time is very similar to lead time, except it doesn't include the time the task waits before being addressed by developers; it only includes the time from when the task is started and finished" [4]. Measuring both lead time and cycle time is essential in a team's development process as it gauges their ability to deliver quality products by analyzing sections of their process that cause either a reduction or an increase in completion time.

"[Velocity] measures how quickly individual developers can finish a given task" [4] and the rate at which a team completes user stories, software features from a customer's perspective, throughout a sprint. Measuring velocity assists in ensuring a sustainable pace in the development process. A decrease in velocity typically signifies the necessity for a review of the development process to pinpoint and amend the cause of the decline.

B. QA Software Quality Metrics: Test Coverage and Code Coverage

Software Quality Assurance is the process in software development that examines a product's quality in regard to functionality, durability, design, security, and reliability. Its focus is on maximizing these product qualities to ensure customer satisfaction and an efficient software development process, ergo it is beneficial to both developer teams and consumers. In addition to providing high quality software, a reliable quality assurance process provides customers with reasonable expectations for the product. To promote a credible quality assurance process, the following metrics should be measured: test coverage and code coverage.

Test coverage is a metric that measures how thorough testing probes a project [1]. Rigorous test coverage entails using both unit tests, tests that focus on functionality, and user tests, tests that focus on fulfilling user demands regarding usability of an application, that encompass every feature in a software product to expose potential defects and rectify code failures. Test coverage is essential in assisting a team in detecting software bugs before users do and, therefore, is a key metric in quality assurance.

"Code coverage is similar to test coverage, except that it measures the amount of the codebase that is tested by unit tests" [1]. To ensure a lower probability of a codebase having hidden software defects, a high percentage of code coverage is required, since code coverage entails how much of the codebase is covered by unit tests. Consequently, this metric is helpful in monitoring software automated tests to promote efficiency in the quality assurance process.

C. Production Metrics: Active days, Productivity, and Code Churn

Although measuring a team's work efficiency can prove to be a challenging task, it is a key component in fostering effective management for the betterment of project outcomes. Production metrics analyze how developers manage their time in terms of their efficiency and how often they require to review their code for error detection. This information helps assess a developer's skill, allowing management to designate appropriate tasks to developers in an effort to increase efficiency. The following metrics are required to gain information about a team's production: active days, productivity, and code churn.

"Active days simply measures the number of days developers spend actively coding" [1], excluding the planning process of the project. This metric helps determine whether a developer is inefficiently distributing project work hours between a user story and unnecessary activities. To prevent inefficiency, active days should be measured and considered to guide developers in improving their time management capabilities.

Productivity tends to be a difficult metric to measure in professional work environments. For the purpose of this paper, productivity will be defined as the amount of code that is written by a developer. High levels of productivity are required to deliver a product at a sustainable pace, and therefore its measurement can aid in determining the size of a developer team needed to accomplish the goal of a project. Additionally, this metric reveals the conditions in which a developer works best by highlighting moments that result in high levels of developer output.

"Code churn refers to the amount of code that is changed in a piece of software" [5]. Although changes in code are to be expected throughout a software development process, high levels of code churn denote a significant amount of software bugs, in addition to a potentially increased number of undetected defects. Therefore, lower levels of code churn are associated with better software quality.

D. Object-Oriented Metrics: Complexity, Coupling, Cohesion, and Size Metrics, Weighted Methods per Class, Response for a Class, Lack of Cohesion of Methods, Coupling Between Object Classes, Depth of Inheritance Tree, Number of Children, Method Hiding Factor, Attribute Hiding Factor, Method Inheritance Factor, Attribute Inheritance Factor, Polymorphism Factor, Coupling Factor

Object-oriented metrics, as opposed to those used in the conventional functional decomposition and data analysis design approach, need to be able to zero in on the interplay between function and data as a single unit. Measuring a software quality attribute allowed for an assessment of the metric's value as a quantitative measure of software quality. However, the chosen metrics have broad applicability in models.

The metrics for object-oriented systems are separated into two sections, with three traditional metrics and six new metrics developed specifically for object-oriented systems. The first three metrics in Table I. demonstrate how conventional metrics can be applied to the object-oriented structure of methods, as opposed to functions or procedures[6].

TABLE I. SOFTWARE QUALITY METRICS FOR OBJECT ORIENTED TECHNOLOGY[6]

Type	Metric	Application
Traditional	CC (Cyclomatic Complexity)	Method
Traditional	Size (Line of Code)	Method
Traditional	COM (Comment percentage)	Method
Object-Oriented	WMC (Weighted Methods per Class)	Class/Method
Object-Oriented	RFC (Response For a Class)	Class/Message
Object-Oriented	LCOM (Lack of Cohesion of Methods)	Class/Cohesion
Object-Oriented	CBO (Coupling Between Objects)	Coupling
Object-Oriented	DIT (Depth of Inheritance Tree)	Inheritance
Object-Oriented	NOC (No. of Children)	Inheritance

McCabe's cyclomatic complexity is frequently used to evaluate the difficulty of an algorithm. In the majority of situations, a method with a lower cyclomatic complexity is preferable, even if it necessitates delaying decisions through message passing. Due to inheritance, cyclomatic complexity cannot be utilized to assess the difficulty of a class. However, it can be utilized in conjunction with other metrics to provide a more accurate evaluation of the class's complexity [6].

The size of a method or routine can be quantified using both physical dimensions and the number of lines of code. It indicates how well the code is written from a programming and maintenance perspective. Larger procedures pose a greater threat in terms of Understandability, Reusability, and Maintainability given that size influences the ease with which a procedure can be comprehended [6].

Counting comments can be added to the line counts used to calculate the Size metric, both inline and outside of the code. The comment percentage is calculated by subtracting lines of code from blank lines. The SATC recommends a percentage of comments of around 30%. Since comments are useful to developers and maintainers, this metric is used to assess the qualities of Understandability, Reusability, and Maintainability [7].

The WMC is the number of methods in a class or the sum of their complexities (as measured by cyclomatic complexity). The second metric is difficult to implement because, due to inheritance, not all methods are visible within the class hierarchy. The amount of time and effort required to create and maintain a class can be roughly estimated by looking at the number of methods and the complexity of those methods. Because children inherit all of the methods defined in a class, the potential impact grows as the number of methods in the parent class grows. Classes with many methods are frequently more tailored to a specific use case, reducing their reusability. The ease with which something can be understood, maintained, and reused is measured [6], [7].

The RFC is the concrete embodiment of all methods that can be called when a message is received by an object of the class or when a method within the class is invoked. This includes all methods at the class level. This criterion considers both a class's internal complexity (as measured by the number of methods) and its external interactions with other classes. The number of methods that can be called via messages increases the complexity of a class. When many methods can be called in response to a message, testing and debugging the class becomes more difficult because the tester must understand the code more thoroughly. Knowing the absolute worst response value makes it easier to allocate testing time effectively. The ease with which the system can be explained, maintained, and tested is measured [6].

Similarities between procedures can be evaluated with LCOM by comparing their input variables and attributes (class structure). Separation of methods measures, however crude, are useful for finding poorly designed classes. Cohesion can be evaluated in at least two distinct ways [6], [7]:

First, determine the percentage of methods in a class that make use of each data field in the class. Take the overall percentage and deduct it from 100. The lower the percentage, the more in sync the class's information and methods are.

Secondly, if two methods work on the same attributes, they are more analogous to one another. Determine how many unique sets resulted from combining the attribute sets employed by the various approaches.

The cohesion of the classes seems to be strong. Development errors are more likely to occur when there is a lack of cohesion or low cohesion. Subdividing classes with low cohesion into two or more subclasses with higher cohesion is likely. Effectiveness and Reusability are measured by this parameter.

How many other classes is this one coupled with? That is what the CBO metric looks at. The degree of interdependence of a class is measured by counting the number of separate class hierarchies it employs that are unrelated by inheritance. Excessive coupling limits the possibility of reuse in a modular design. The more modular a class is, the easier it is to implement in another context. The design becomes more fragile and difficult to maintain as the number of couples increases. A strongly coupled module makes the system more difficult to understand, change, and correct due to its interdependence with other modules. Building systems with minimal interdependence is one way to simplify them. Encapsulation is encouraged and modularity is enhanced. The CBO examines efficiency and reusability [6].

The DIT, or the number of parent classes, quantifies how far back in the inheritance tree a given class can be traced. Classes lower in a hierarchy tend to inherit more methods, making it more difficult to predict how they will behave. More methods and classes in a deeper tree indicate greater design complexity, but also more opportunities to reuse inherited functionality. DIT's success can be measured in part by how well it inherits techniques from previous eras (NMI). This is an efficiency and reusability metric with links to readability and testability [6], [7].

The NOC, or child count, of a class equals the number of subclasses that are directly subordinate to it. It is a measure of a category's potential weight in shaping the system's architecture and operation. When there are many offspring, the subclassing technique may be abused, resulting in improper abstraction of the parent. Because inheritance is a form of reuse, the greater the number of children, the greater the potential for reuse. If a class has a large number of children, more time may be required to test the methods. As a result, the primary metrics considered by NOC are effectiveness, reusability, and testability [7].

E. Traditional Function-Oriented Metrics: Line of Code, Token Count

Measuring function-oriented metrics is another method used to evaluate software quality. Function-oriented metrics are defined as the number of functionalities in a software system, but, since functionality isn't a component that can be directly calculated, these metrics focus on the countable functionalities. The following metrics can be used to determine software functionality information: line of code, token count, and software science.

The line of code metric is one of the simplest ways to assist in evaluating software quality. It is based on counting the number of lines/statements in a program/software according to counting rules depending upon the programming language used to write the program [2]. It is closely related to the productivity metric mentioned earlier since productivity is defined as the amount of code that is written by a developer. Although it is an easy metric to measure, its simplicity exposes its flaw of assigning each line the same level of complexity, ignoring the function of each line in the program. Nevertheless, it is an essential aspect in calculating an application's size and its effect on software quality.

Token size metrics are used to determine the number of tokens, defined as either operators or operands, in a program. Variables and constants are classified as operands, whereas commands, methods/functions, or arithmetic operators are classified as operators. Measuring token size metrics is yet another form of evaluating software quality by calculating the size of a program [8].

F. Component-Oriented Metrics

Software engineering encompasses a plethora of branches. One such branch is component-based software engineering whose focal point is on the reuse of software components throughout a system in terms of its design and development. A software component is defined as a module envelops information regarding program functionality. The main goal for component-based software engineering is to remedy the issue with having explicitly defined objects that have limited applicability by shifting the focus to the development of reusable program parts. The following component-oriented metrics should be measured to help assess the software quality of a system: reusability, coupling, and component cohesion.

Since component-based software engineering focuses on reusing software components to build systems, reusability is a key component in dictating whether a system built utilizing a component-oriented approach achieves high software quality. Reusability of a component is defined as the probability that a component can be reused to build software [2].

Coupling and component cohesion, as mentioned in the software quality metrics for object-oriented technology, are additional metrics that assess a component-based system's software quality. Coupling refers to the relationship between software components, whereas cohesion measures the interdependence within components. An ideal component-oriented approach would implement a system with low coupling and high cohesion to increase software quality[8].

G. Aspect-Based Metrics: Number of Aspects, Number of Pointcuts per Aspect, Number of Advices per Aspect, Degree of Crosscutting per Pointcut, Response for Advice

A software system can be thought of as a collection of related modules that work together to model a problem. Although the object-oriented paradigm (classes, objects, methods, and attributes) provides abstractions, they are frequently insufficient to represent all of a software system's concerns. A "crosscutting concern" is encapsulated by a "aspect" in Aspect-Oriented Programming, and AOP also makes it easier to compose aspects and components such as classes, methods, and attributes[1], [9].

The increasing complexity of software necessitates the adoption of new development paradigms. The goal of aspect-oriented programming is to solve problems that traditional object-oriented programming cannot. Engineers in the field of software development would have to develop novel metrics and models to assess software quality in the face of such a paradigm shift [9].

In order to validate claims made about the robustness, reusability, and maintainability of software systems built with this new paradigm, software metrics for aspect-oriented systems are essential. Despite the fact that the Aspect-Oriented paradigm is still in its infancy, a number of researchers have already reported on its performance. Some examples are given here [9]:

1. The "number of aspects" refers to the total number of features that have been built into the software by its developer, and this value is equal to that tally.
2. The NPA, or the number of pointcuts used in a single aspect, is the total number of pointcuts used in a single aspect.
3. The term "Number of Advices per Aspect" (NAA) is used to describe the total number of pieces of guidance used to construct a single aspect.
4. DCP stands for "degree of crosscutting per pointcut," which measures how many different classes a given pointcut within an aspect affects.
5. Response for Advice (RAD) is the process of keeping track of how many solutions have been implemented after reading a recommendation.

III. Software Quality Prediction Techniques

Since certain defects or failures in such high-assurance systems can have disastrous consequences, one effective method for improving the quality and reliability of a high-assurance software project/product is to detect and correct software defects (bugs) early enough during the software development process and prior to system deployment and operation. Several methods and strategies have been developed for this function. Software quality modelling is one of these methods, and it has a lot going for it. To determine whether a specific instance of a program module (fp) belongs to the fault-prone (fp) or not fault-prone (nfp) class, software quality assurance professionals often use data mining techniques to analyze software metrics (attributes or features) collected during the software development process[10].

Several researchers have focused on improving the quality of the final software product, which falls under the umbrella

of software engineering. The testing team benefits from fault prediction and early identification because it enables them to focus their testing efforts on the modules most likely to develop issues. Typically, structural measurements of software (source code metrics also known as SCMs)—including dimensions such as size, cohesion, coupling, complexity, and inheritance—are employed to construct models that forecast software quality [11].

A. Software Quality Prediction Using Machine Learning: Decision Trees

The field of machine learning in computer science is a burgeoning sector with continuous developments being made on a regular basis. It utilizes artificial intelligence to analyze data and improve performance with minimal user involvement. Machine learning is beneficial because it provides organizations methods of quickly analyzing data in order to predict business ecosystem changes, allowing companies to stay prepared on future customer trends. Due to its prowess in prediction, machine learning has been regarded as one of the most efficient methods of software quality prediction. It provides various techniques for software quality prediction such as: decision trees, Bayesian classification, rule-based classification, nearest neighbors, and more. For this paper, we are going to be focusing on decision trees[10].

Machine learning implements decision trees to all possible paths in decision-making. Decision trees utilize a top-down approach in which the top of the tree is its root and each connected node is a possible decision. The edges connecting nodes represent possible paths from each decision. Decision trees can be split into two categories[12]: classification trees and regression trees.

Both classification and regression trees are built through an iterative process of splitting data into partitions, then separating the data further into branches with each leaf representing a decision. Classification trees are implemented when decision results are split into binary classes, whereas regression trees are implemented whenever decisions represent continuous numerical results.

One of the advantages that come from implementing decision trees is simplicity. Visualization of the data is made intelligible when utilizing decision trees, hence different members of an organization are able to understand the model's output. Furthermore, since decision trees split data into either binary/categorical or numerical data, converting data values of numeric columns to be on a similar scale for the purpose of normalization is not required [12].

Alternatively, there are also disadvantages to implementing decision trees. Results from decision trees tend to be inaccurate when considering unwanted behaviors in the dataset—an occurrence known as overfitting. Additionally, if the dataset is too large then the tree results in having too many nodes, leading to complexity and contributing to overfitting [12].

B. Software Quality Prediction Using Feature Selection

Effective feature selection methods can significantly improve a machine learning model's predictive power and speed. Feature selection (FS) is the process of selecting a subset of relevant features to maintain or improve the accuracy of prediction models. Data sampling is used to modify the dataset to change its balance level in order to solve the problem of traditional classification models being biased toward the over-represented (majority) class. Recent research confirms the effectiveness of boosting (building multiple models, each tuned to perform better on instances misclassified by previous models) in resolving the class imbalance problem [13].

When tackling the issue of a high-dimensional dataset, feature selection is a crucial first step in the data pre-processing phase. This issue is compounded by redundant variables, in which one independent variable is highly correlated with another and can be eliminated, and irrelevant variables, in which one or more features (independent variables) have no effect on the target features (dependent variables) [13].

The filter method, which identifies the features without building machine learning models using heuristically determined relevant knowledge, the wrapper method, which integrates features into a prediction model to select relevant features, the embedded method, which takes advantage of both filter and wrapper methods and selects the best subset during the creation of the model, and the ensemble method, which uses multiple models in combination to select the best subset of features. In Table II, we see that each strategy makes use of a number of distinct methods [14].

TABLE II. DESCRIPTION OF THE FEATURE SELECTION TECHNIQUES[14]

Name	Description
Relief	Relief uses an iterative sampling method to evaluate each attribute and calculate a score for each. In the Relief algorithm, the total score is calculated as the difference between the attribute values of any two neighbours.
Symmetrical uncertainty	Attributes are picked using symmetrical uncertainty, which addresses the issue of information gain bias by computing the symmetrical uncertainty. For this issue, we need attributes with multiple values, normalized between zero and one.
Gain ratio	The gain ratio uses a decision tree whose size and number of branches determines the attributes chosen. In order to reduce the information gain's inherent bias during splitting, a new measure called the "gain ratio" was developed.
Information gain	The amount of data generated by feature items can be estimated using the concept of "information gain." Without taking into account any of the intrinsic information, it employs the same concept of gain ratio.
Chi-square	When solving a classification problem, chi-square is calculated as the correlation between each attribute and the dependent variable (i.e., the target) (i.e. categorical feature). The Chi-square test determines whether or not the correlation between these sample categorical characteristics and the population correlation is significant.
Correlation based	The entire set of attributes is used in correlation-based feature selection, and then the attributes are ranked by their level of correlation to the dependent variable (i.e. target).
Best first	When using best first, one can either start with a group of attributes that is completely empty and work their way forward, or start with a group of attributes that is completely full and work their way backward.
Support vector machine	In the wrapper technique, feature selection can be carried out with the aid of a support vector machine model.
Kolmogorov Smirnov	The Kolmogorov-Smirnov statistic is used to make attribute selections. It measures how far apart each class's empirical distribution function is from the other classes' distribution functions to determine how well an attribute distinguishes between them.

Greedy	For each attribute, you can use greedy for a forward or a backward search. It keeps going until introducing or removing attributes has a negative impact on the prediction accuracy.
Rough sets	Rough Sets is defined as a formal approximation of a regular set. A theory based on the classical set is used.

To determine the effect of feature selection methods on prediction accuracy, it was found that all of the feature selection methods led to better prediction accuracy compared to using all features (i.e., without using feature selection), and it was reported that the ensemble method achieved the best prediction accuracy among the other feature selection methods used in this study [14].

IV. Conclusion

Software defects are detrimental in all areas of the software development process. They create code conflicts and delays in production, consequently diminishing customer satisfaction. Software quality metrics have been developed to assist in rectifying these errors. In this paper we gave an overview of these metrics and their importance in establishing properly-maintained and bug-free applications by analyzing code size, complexity, and reliability. In addition to software quality metrics, we detailed two specific software quality prediction techniques—decision trees in machine learning and feature selection.

To improve customer satisfaction, it is essential to deliver a product that is high in quality, ergo it is necessary to place focus on functionality, reliability, and security. Software quality metrics specify areas of improvement in the software process, whereas software quality prediction facilitates identifying software defects early in the software development process to assess the quality of a product. Ultimately, measuring these metrics and implementing these prediction techniques are beneficial to developing software that is free of defects and high quality.

V. References

[1] P. Ward, “Software Quality Metrics Explained With Examples | NanoGlobals,” Oct. 01, 2022. https://nanoglobals.com/glossary/software-quality-metrics/ (accessed Oct. 02, 2022).

[2] R. S. Chhillar and S. Gahlot, “An Evolution of Software Metrics: A Review,” in Proceedings of the International Conference on Advances in Image Processing, 2017, pp. 139–143. doi: 10.1145/3133264.3133297.

[3] “IEEE Standard for a Software Quality Metrics Methodology,” IEEE Std 1061-1992, pp. 1–96, 1993, doi: 10.1109/IEEESTD.1993.115124.

[4] M. Maddox and S. Walker, “Agile Software Quality Metrics,” in 2021 IEEE MetroCon, 2021, pp. 1–3.

[5] “Survey on impact of software metrics on software quality”.

[6] S. Parthasara and . N. A., “Analyzing the Software Quality Metrics for Object Oriented Technology,” Information Technology Journal, vol. 5, no. 6, pp. 1053–1057, Oct. 2006, doi: 10.3923/itj.2006.1053.1057.

[7] L. H. Rosenberg and L. E. Hyatt, “Software Quality Metrics for Object-Oriented Environments,” 2002.

[8] M. S. Rawat, A. Mittal, and S. K. Dubey, “Survey on impact of software metrics on software quality,” IJACSA) International Journal of Advanced Computer Science and Applications, vol. 3, no. 1, 2012.

[9] A. Singh and L. Balani, “Software Quality Metrics for Aspect-Oriented Programming,” Nov. 2015.

[10] S. Shafi, S. M. Hassan, A. Arshaq, M. J. Khan, and S. Shamail, “Software quality prediction techniques: A comparative analysis,” in 2008 4th International Conference on Emerging Technologies, 2008, pp. 242–246. doi: 10.1109/ICET.2008.4777508.

[11] S. Bouktif, F. Ahmed, I. Khalil, and G. Antoniol, “A novel composite model approach to improve software quality prediction,” Inf Softw Technol, vol. 52, no. 12, pp. 1298–1311, 2010, doi: https://doi.org/10.1016/j.infsof.2010.07.003.

[12] F. Alaswad and E. Poovammal, “Software quality prediction using machine learning,” Mater Today Proc, vol. 62, pp. 4714–4720, 2022, doi: https://doi.org/10.1016/j.matpr.2022.03.165.

[13] Z. Xu, J. Liu, Z. Yang, G. An, and X. Jia, “The Impact of Feature Selection on Defect Prediction Performance: An Empirical Comparison,” in 2016 IEEE 27th International Symposium on Software Reliability Engineering (ISSRE), 2016, pp. 309–320. doi: 10.1109/ISSRE.2016.13.

[14] H. Alsolai and M. Roper, “A systematic review of feature selection techniques in software quality prediction,” in 2019 International Conference on Electrical and Computing Technologies and Applications (ICECTA), 2019, pp. 1–5.