Unveiling Inefficiencies in Open-Source Code Using Multistage Analysis with Software Metrics

Rasheed Mohammed Al-Nofah[1], Muhammad Shumail Naveed[1]
[1]Department of Computer Science & Information Technology, University of Baluchistan Quetta, Pakistan
***Correspondence:** Rasheed Mohammed Al-Nofah, alnofahr@gmail.com

Citation | Al-Nofah. R. M, Naveed. M. S, "Unveiling Inefficiencies in Open-Source Code Using Multistage Analysis with Software Metrics", IJIST, Vol. 5 Issue. 4, pp 360-370, Oct 2023
Received | Sep 14, 2023**; Revised |** Sep 24, 2023**; Accepted |** Sep 28, 2023; **Published |** Oct 02, 2023.

Software development is challenging due to its technical complexity and time-consuming nature. To overcome these difficulties, various technical solutions have been introduced. In commercial software development, code repositories serve as valuable resources, reducing the time and cost involved in the process. The utilization of pre-developed open code repositories has proven to reduce development time. However, ample amount of work has not determined whether these repositories are testable, maintainable, free of dead code, and have a concise implementation of equivalent algorithms. The objective of this article is to address this gap by thoroughly analyzing the complexity and maintainability of code repositories, determining the impact of removing dead code on size, complexity, and maintainability. For this study, a total of 200 Python open-source code were analyzed using RADON, a widely-used metric tool for assessing cyclomatic complexity, size, volume, and maintainability. The identification of dead code within the repositories was accomplished using Vulture, supplemented by expert evaluation. It has been revealed that the majority of the examined code included dead code, and the removal of this code led to a significant reduction in cyclomatic complexity, volume, and size, while improving code maintainability, as observed by the Mann Whitney U test. The study concludes that the blind use of open-source code is not safe. It strongly recommends the community to thoroughly explore and examine such code from different perspectives before actual implementation. The novelty of this study lies in the use of multiple software metrics in a multi-stage analysis to examine the impact of removing dead code on program complexity, size, and maintainability.

Keyword: Software Development, Code Maintenance, Open-source code, Cyclomatic Complexity, Maintainability Index.

Introduction:

Computing technology has become ubiquitous in various manifestations, including computers, tablets, smartphones, the internet, cloud computing, email, text messages, social media, and more. It bears great significance in the lives and careers of an increasing number of individuals [1]. Computational methods are employed in education to enrich the learning experience, as well as in economic, political, and other diverse domains.

Software constitutes a fundamental element within the realm of computing technology. The application of software assumes a pivotal function in automating organizational responsibilities and in providing updates on the advancement or potential setbacks in the organization's undertakings. This dynamic greatly bolsters the efficiency and efficacy of the company's operations. Implementing software results in a decreased workload and expedites the automation of a multitude of tasks. Additionally, it aids in the reduction of human errors, thereby elevating overall efficiency and uniformity [2].

Software is crafted within a structured framework known as programming languages. Computer programming stands as a cornerstone of computer science and, as such, represents a crucial competence for learners to attain [3]. The importance and ubiquity of software across nearly every domain have spurred the ongoing evolution and progress of programming languages, consequently augmenting the demand for proficient software developers. It is projected that by 2026, there will be a substantial surge in job openings for software developers [4]. This rapid expansion in programming roles accentuates the challenges tied to formulating programming languages. As time has passed, noteworthy strides have been taken, resulting in the emergence of a plethora of programming languages [5]. Among the most widely adopted and prevalent programming languages are C, C++, Java, and Python.

C has emerged as a programming language of significant relevance in the contemporary computing landscape [6]. It stands as a foundational technology and remains a favored choice for introducing programming concepts [7], rendering it especially suitable for engineering applications. Throughout its history, a multitude of programs have been composed using the C language. Despite the proliferation of various alternative programming languages, C has consistently upheld its popularity.

C++, crafted by Bjarne Stroustrup, is a versatile programming language designed to accommodate a wide spectrum of applications [8]. It finds particular favor in contexts where performance and efficient resource utilization take precedence. C++ is fundamentally categorized as an imperative language, treating a program as a series of statements that alter the program's state. Beyond its imperative nature, C++ also embodies procedural aspects, providing support for procedures and subroutines. Moreover, it embraces the tenets of structured programming and encompasses features of static typing, robustly supporting both object-oriented and generic programming paradigms. Significantly, C++ furnishes extensive capabilities for manipulating low-level memory.

Java, developed by James Gosling [9], stands as a remarkably successful and influential programming language that holds a distinguished position within the realm of programming languages. It excels across multiple dimensions, encompassing lexical, syntactic, semantic, and pragmatic facets, thereby serving as an exemplar for the development of subsequent programming languages. Technically, Java is defined by its robust and static typing [10]. It operates as a hybrid of compiled and interpreted language paradigms. Among its notable attributes are platform independence and formidable security features.

Python stands as one of the most widely embraced programming languages globally. Crafted by Guido van Rossum in 1989, Python has emerged as the preeminent choice for applications in vital and burgeoning domains like natural language processing and big data analytics. It effectively bridges a significant divide between high-level applications akin to

spreadsheets and statistical analysis tools, and languages oriented towards system development such as C, C++, and Java [11].

Proficiency in programming languages is an indispensable skill for computer science experts. However, the instruction and acquisition of programming languages prove challenging due to the intricate interplay of grasping theoretical foundations, effectively applying semantic and syntactic coding principles, and honing algorithmic aptitude [12].

The development of software demands significant skills, expertise, and time, rendering this process costly, time-consuming, intricate, and complex. In response to these challenges, numerous solutions have been introduced to streamline software development, with one prevalent approach being the utilization of open code repositories The applicability of these repositories spans across various domains within the realm of Empirical Software Engineering [13].

Open-source repositories have garnered substantial utilization in the software industry, significantly expediting the software development pace. These repositories fulfill a pivotal function by presenting an expansive assortment of openly accessible code, libraries, and frameworks. By harnessing these resources, developers can expedite their software development endeavors. The incorporation of open-source repositories serves to streamline development initiatives, mitigating the necessity to initiate projects from the ground up and empowering developers to construct upon pre-existing solutions. This approach not only economizes time but also fosters collaborative efforts and nurtures innovation within the software development community.

There exist several fundamental attributes inherent to software projects, such as maintainability [14][15] and testability [16][17], which inherently impact them. Open-source repositories alleviate software development efforts, inherently diminishing both development time and costs. Nevertheless, it remains uncertain whether these repositories possess simplicity in terms of testability, just as it is unknown whether they exhibit maintainability.

Cyclomatic complexity is a metric for evaluating the testability of a program, calculated by determining its logically independent pathways. The maintainability represents the ease with which a software system or component can be modified to rectify faults and enhance performance. The Maintainability Index (MI) is a measurement used to track maintainability.

Numerous studies have been conducted on open code repositories, with a special emphasis on vulnerabilities. However, there has been a lack of significant analysis concerning the complexity and maintainability of these open code repositories. GitHub serves as a vast reservoir of open-source repositories, providing users with the option to "star" these code repositories. These stars serve as tokens of appreciation and indicators of popularity within the GitHub ecosystem. Despite this, it is important to note that these repositories may exhibit varying degrees of quality and could potentially harbor vulnerabilities that could be exploited by malicious hackers. A comprehensive study [18], delved into the relationship between the number of stars associated with GitHub's code repositories and the presence of vulnerabilities within their code. This investigation employed a static code analyzer to meticulously scrutinize the vulnerabilities present in ten widely recognized C++ source repositories on GitHub.

Remarkably, the examination revealed a staggering total of 3487 vulnerabilities within the dataset. Strikingly, not a single repository in the dataset remained untarnished by flaws. Subsequent analysis involving a statistical examination illuminated a notable difference among the different code repositories within the dataset in terms of detected vulnerabilities. A correlation coefficient test, however, failed to identify any significant correlation between a repository's star count and the frequency of vulnerabilities. This implies that the heightened popularity of a code repository on GitHub, as gauged by the accumulation of numerous stars, does not inherently reflect its level of security integrity.

The investigation presented in study [19] explored the security-related vulnerabilities inherent in programming languages by uncovering variations among them within widely-used code repositories. This study meticulously examined 708 programs based on severity-based guidelines. Through this comprehensive analysis, a total of 1371 instances of vulnerable code were unveiled. Among these, 327 were linked to the C language, 51 to C++, and 993 to Java. The statistical analysis underscored the significant difference among these language-specific vulnerabilities.

An empirical analysis [20] was undertaken to delve into the Open-Source development process, specifically from the perspective of developer engagement within the production cycle. The study intricately examined how developers contribute to projects, considering factors such as their level of involvement, the scale of their contributions, and the nature of those contributions. A dataset encompassing 53 Open-Source projects was meticulously gathered, spanning various application domains. This collection of data encompassed crucial variables, including developer headcounts, patterns of code modifications, and the evolutionary trajectory of project size and complexity over time. The results of the investigation compellingly highlight the existence of recurring patterns within Open-Source software development. Remarkably, these patterns proved to be universal across all projects under consideration. This consistency persisted despite the absence of standardized development processes, the diverse range of application domains, and the dispersed global contributions from individuals.

The study [21] examined the correlation between the transparency of a publication, as denoted by the characteristics of its open-source repository, and its scientific influence. Through the utilization of the Mann-Whitney test and Cliff's delta, a statistically significant distinction in citation counts emerged when comparing papers with and without an associated open-source repository. The study also revealed a significant statistical correlation between citation counts and various features of repository interaction. These features encompassed metrics such as Stars, Forks, Subscribers, and Issues.

The growing popularity of third-party repositories renders them an appealing focal point for software supply chain attacks. Attackers have been observed to manipulate genuine packages by infusing them with malicious code, consequently amassing over 100,000 downloads of the compromised packages. To counteract this, a study [22], has introduced the idea of employing source code repositories to identify illicit insertions within a package's distributed components. The initial assessment effectively showcased the viability of the proposed method in detecting known attacks, especially when malevolent code was inserted into PyPI packages. An in-depth scrutiny of 2666 software artifacts substantiates the notion that this technique proves apt for conducting lightweight analyses on real-world packages.

A study [23] focusing on open-source repositories introduced the SonarCloud Vulnerable Code Prospector for C. The primary objective was to gather vulnerable source code instances from open-source repositories associated with SonarCloud, an online tool designed for conducting static analysis and identifying potentially vulnerable code segments. This tool identifies and tags files that potentially contain vulnerabilities, offering a curated collection of tagged files suitable for feature extraction. These files serve as valuable resources for constructing training datasets to train Machine Learning algorithms. This study presented a comprehensive descriptive analysis of the aforementioned files, offering an overview of the current status of vulnerabilities within C programming, specifically addressing issues like buffer overflows, as observed in the examined public repositories. The study's findings highlight that buffer overflow vulnerabilities have remained a focus of investigation for many years, underlining that establishing preventive measures and defense mechanisms against such vulnerabilities remains a fundamental pillar of cybersecurity endeavors.

It is widely noted that open-source code often experiences subpar code quality. In order to investigate this issue, a study [24] analyzed eleven open-source software projects. The goal was to assess whether the quantity of contributing developers has an impact on code quality. This assessment was conducted using surrogate measures of code quality such as cyclomatic complexity, lines of code per function, comment density, and maximum nesting. The study did not find significant evidence to support the notion that the number of contributing developer's influences software quality.

Objectives

The purpose of this study is to investigate the complexity and maintainability of open-source code. It also seeks to examine whether code repositories are devoid of dead code. Furthermore, the study will explore the possibility of enhancing program testability and maintainability by eliminating potentially dead code from the code.

Material and Methods:

This study aims to assess the testability and maintainability quality of open code repositories, as well as their potential for containing dead code. Additionally, it investigates whether removing potential dead code from these repositories could enhance program testability and maintainability. The methodology followed during the study is shown in Figure 1.

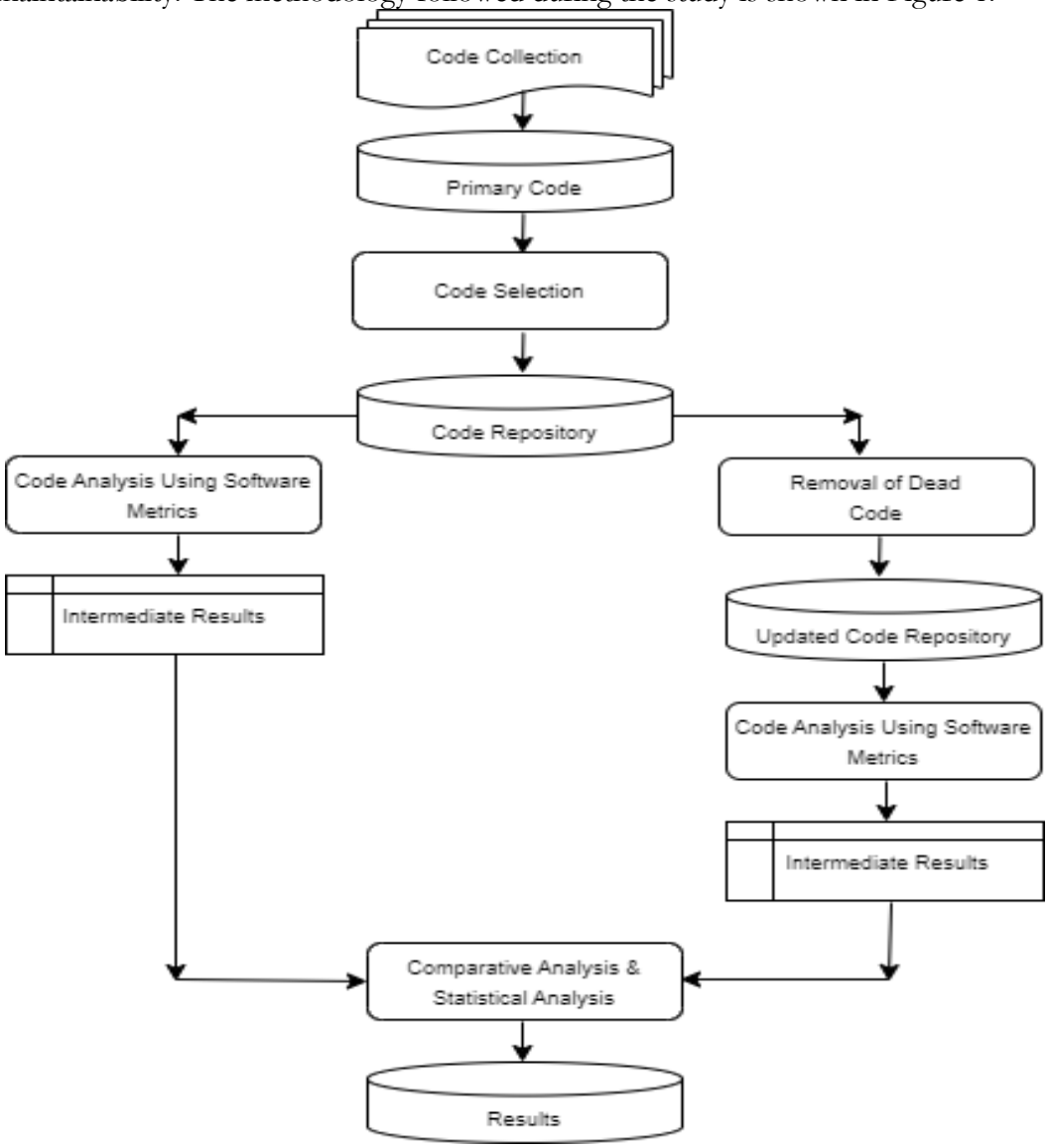

Figure 1. Research Methodology

The study encompassed the collection of open-source materials from various platforms, such as GitHub, Source Forge, and Bit Bucket. or the study, primary data (source code) was downloaded from popular online sources, including GitHub, Bitbucket, and Gitlab. The code in these repositories is available to all users and, therefore, applicable for other studies. Initially, a total of 287 open-source code samples were gathered and organized into a primary code collection between December 2022 and February 2023. Following consultation with two programming experts, a subset of 200 open-source code samples was selected for the study, and a dedicated code repository was created for them. Subsequently, this chosen code underwent analysis in two distinct phases.

In the first phase, critical metrics including cyclomatic complexity, Halstead volume, raw size, and maintainability index were computed using RADON, a robust Python-based tool. This phase of analysis yielded valuable insights into the structural and operational characteristics of the code.

McCabe's cyclomatic complexity stands as a standard metric for assessing software complexity, quantifying it in relation to the count of linear independent paths [25][26]. The metric of cyclomatic complexity is consistently referenced as a valuable predictor for various software attributes, including reliability and development effort. The cyclomatic complexity of a program (p), represented as a graph (G), is calculated as [27].

$$p(G) = Edges - Nodes + 1$$

The cyclomatic complexity metric is used to identify code sections that will be hard to maintain or debug. This method is based on calculating the total number of logical independent paths and the presence of selection and repetition statements [28].

The Halstead complexity metric is a crucial method used to measure the complexity of program code [29]. The Halstead volume is a way to scale the size of the implementation for any given algorithm [5]. The calculation of the Halstead volume is as follows.

$$V = N \times Log2\ (n),$$

Where,

$$N = \text{total number of operators and operands}$$

$$n = \text{program vocabulary}$$

The Maintainability Index (MI) is a quantitative software metric that ensures reliability [30]. The MI is a value used to estimate the maintainability of code. It is calculated using lines of code, Cyclomatic complexity, and the Halstead metrics. The standard formula for MI:

$$MI = 171 - 5.2 * \ln(V) - 0.23 * (CC) - 16.2 * \ln(LOC)$$

Where,

V = volume of Halstead complexity metrics

CC = Cyclomatic complexity

LOC = Lines of Code

The revised formula for Maintainability Index:

$$MI = MAX(0,(171 - 5.2 * \ln(V) - 0.23 * (CC) - 16.2 * \ln(LOC))*100 / 171)$$

During the second phase of the study, the collected code was scrutinized to identify potential dead code, utilizing Vulture, a powerful tool designed for dead code detection. The findings from Vulture were cross-validated by two human experts. Using the identified findings, the code repository was enhanced by removing the flagged dead code. The resultant code, following the removal of dead code, was then organized into an updated code repository. This updated code repository underwent examination, and once again, metrics like cyclomatic complexity, Halstead volume, raw size, and maintainability index were calculated using RADON in Python (version 3.9).

The results from both phases were subsequently compared and subjected to statistical analysis using SPSS (version 25). To visually represent the results, R (version 4.2.3) was employed.

Results:

The study was conducted in two stages, and the original code was examined by performing computations for cyclomatic complexity, Halstead volume, raw size (source lines of code), and the maintainability index. The results obtained after the analysis are presented in Table 1.

Table 1. Result of Analysis on Original Code

Metrics	Mean	Median	Std. Dev.	Min	Max	Skewness	Kurtosis
Cyclomatic Complexity	3.67	3	2.79	1.00	28.00	4.11	29.23
Halstead Volume	803.24	412	1380.08	4.00	12010.00	5.20	33.88
Raw (SLOC)	298.78	282	214.83	52.00	1688.00	3.13	15.94
Maintainability Index	57.66	57	16.10	11.00	100.00	-0.12	0.26

The clear variation in the collected programs can be observed across all the metrics calculated within the code repository. To offer a clearer visualization of the results, line charts have been created and are displayed in Figure 2.

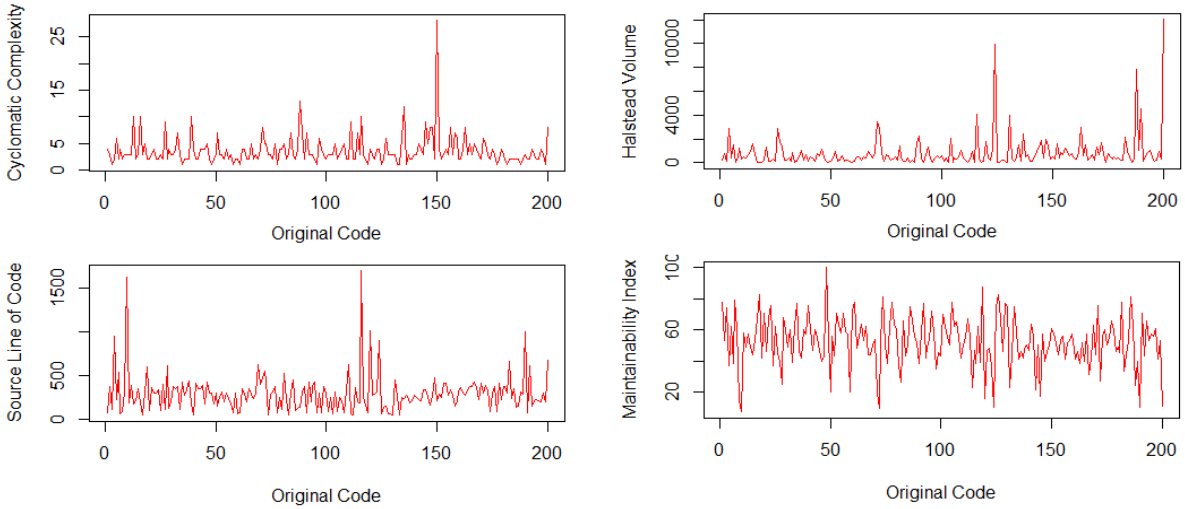

Figure 2. Original Code Results in Line Charts

The second stage of the study was initiated with the identification and removal of dead code with Vulture, carried out in consultation with human experts. The study identified varying sizes of dead code in the analyzed code repositories, and a line chart illustrating the presence of dead code is shown in Figure 3.

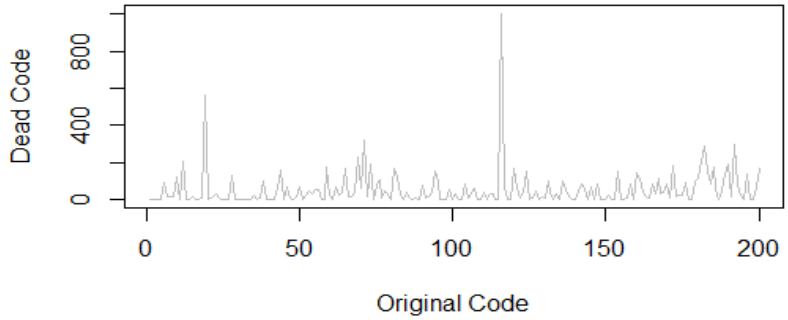

Figure 3. Line Charts for Identified Dead Code

The line charts clearly display the variation of dead code in the analyzed code, indicating the generality of the actual dataset. The improved results were organized into an updated code

repository and further evaluated for cyclomatic complexity, Halstead volume, raw size, and the maintainability index. The results obtained after the analysis are shown in Table 2.

Table 2. Result of Analysis on Improved Code

Metrics	Mean	Median	Std. Dev.	Min	Max	Skewness	Kurtosis
Cyclomatic Complexity	3.64	3	2.74	1.00	28.00	4.27	31.43
Halstead Volume	640.56	326	1118.74	4.00	10979.00	5.66	42.57
Raw size (SLOC)	244.35	222	171.80	19.00	1501.00	2.87	15.71
Maintainability Index	52.30	53	16.40	7.00	100.00	-0.33	0.48

The results obtained after the analysis of the improved code reveal differences compared to the original code in terms of cyclomatic complexity, volume, size, and maintainability index. To provide a clear illustration of these results, line charts have been created and are displayed in Figure 4.

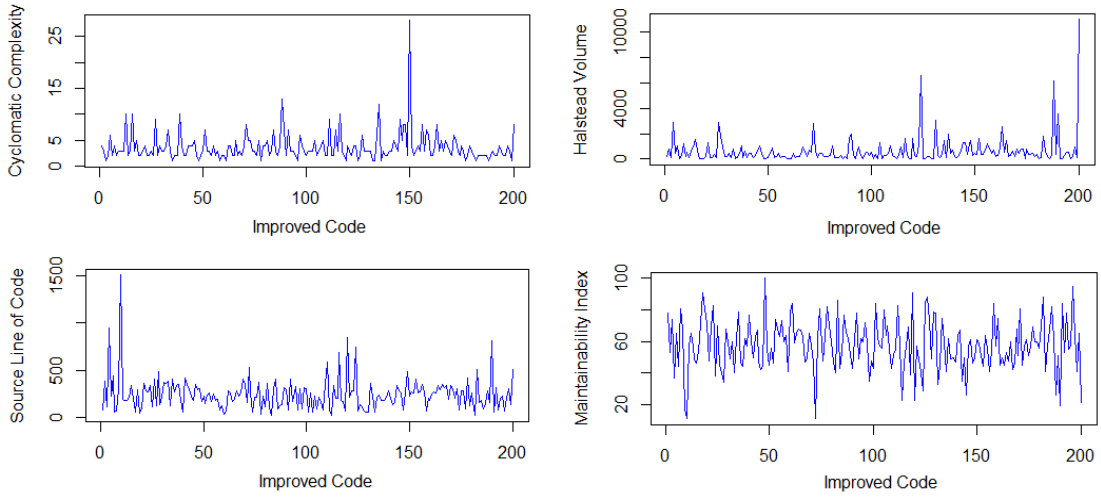

Figure 4. Improved Code Results in Line Charts

The results of the analysis for both the actual code and the improved code were further examined with normality tests, and the findings are presented in Table 3.

Table 3. Result of Normality Tests

Metrics	Code	Kolmogorov-Smirnov			Shapiro-Wilk		
		Statistic	df	Sig.	Statistic	df	Sig.
Cyclomatic Complexity	Original	0.24	200	0.00	0.66	200	0.00
	Improved	0.24	200	0.00	0.67	200	0.00
Halstead Volume	Original	0.28	200	0.00	0.48	200	0.00
	Improved	0.28	200	0.00	0.49	200	0.00
Raw (LOC)	Original	0.12	200	0.00	0.79	200	0.00
	Improved	0.17	200	0.00	0.74	200	0.00
Maintainability Index	Original	0.05	200	0.20	0.99	200	0.13
	Improved	0.07	200	0.03	0.98	200	0.00

The Kolmogorov-Smirnov test and Shapiro-Wilk test identified non-normality in most of the results, except for the maintainability index of the original code in both normality tests, and the maintainability index of the improved code in one normality test. The Mann-Whitney U test is employed for analysis due to the non-normal distribution of the data.

A Mann–Whitney U test revealed a significant difference between the original code and the improved code, for cyclomatic complexity ($U = 19951.50$; $Z = -0.04$; $p = 0.97$), Halstead

volume (U = 17665.50; Z = -2.02; p = 0.04), raw size (U = 16011.50; Z = -3.45; p < .05), and maintainability index (U = 16327.50; Z = -3.18; p < .05).

Discussion:

Open-source code is a widely acknowledged method used in software development to reduce development time, budget, and other scarce resources. On one hand, predeveloped repositories of source code offer numerous advantages; however, on the other hand, this code can be of high complexity, large size, and difficult to maintain. This study analyzed Python open-source code to determine its efficiency, conciseness, and ease of maintenance.

The two-stage analysis of the code revealed that the average cyclomatic complexity of the original code was 3.76, compared to 3.64 for the improved code. As a result, the percentage difference of 3.24% indicates that the testability and cyclomatic complexity of open-source code can be enhanced through the removal of dead code.

The average Halstead volume of the original code was 803.24, compared to 640.56 for the improved code. Their percentage difference of 22.54 clearly indicates that the volume of open-source programs is relatively low. This is attributed to the presence of dead code, and thus, the removal of this dead code could effectively manage program volume.

The average raw size of the original code was 298.78, compared to 244.35 for the improved code. Their percentage difference of 20.04 clearly indicates that the size of open-source programs is relatively higher than that of its corresponding improved code. This suggests that open-source code contains dead code that could be deleted, potentially reducing the overall code size.

The average maintainability index of the original code was 57.66, compared to 52.30 for the improved code. Their percentage difference of 9.75 clearly indicates that the open-source code is difficult to maintain within the scope of the present study and the analyzed dataset.

The overall study concluded that open-source code is a valuable asset for commercial software development. However, the blind use of this code is not useful, as the code may have high complexity and size, which will naturally complicate the software, increase its size, and affect maintainability. A detailed analysis of open-source code, including the removal of dead code, is necessary before its use in software development or in any other application.

The novelty of this study lies in the quantitative analysis of code and results, which will be useful for software engineering and researchers in the future. However, there are several limitations to the present research: i) only 200 source codes were examined in the study, ii) only a few elements were examined during the study, iii) a single programming language was considered for the study. In the future, a larger programming corpus will be examined from multiple perspectives, and similarly, multiple programming languages will be considered for the study.

Conclusion:

Software development is intricate and time-consuming, necessitating technical solutions. In commercial development, online code repositories streamline the process by providing predeveloped open code. However, the uncharted viability of this code in terms of testability, maintainability, dead code elimination, and algorithmic efficiency is addressed in the study through a comprehensive analysis of 200 online code repositories. The two-stage analysis of the code and its comparative study revealed that online code is beneficial, but blind usage in development is unsafe due to potential high complexity, size, and maintenance challenges. Thorough examination of this code, including the removal of dead code, proves valuable as it can lead to reduced complexity, conciseness, and enhanced maintainability.

References:

[1] "INTRODUCTION TO COMPUTER PROGRAMMING (BASIC)." https://www.researchgate.net/publication/317182495_INTRODUCTION_TO_CO

MPUTER_PROGRAMMING_BASIC (accessed Oct. 02, 2023).
[2] "Python current trend applications-an overview." https://www.researchgate.net/publication/344569950_Python_current_trend_applications-an_overview (accessed Oct. 02, 2023).
[3] M. S. Naveed and M. Sarim, "Two-Phase CS0 for Introductory Programming: CS0 for CS1," Proc. Pakistan Acad. Sci. A. Phys. Comput. Sci., vol. 59, no. 1, pp. 59–70, Jun. 2022, doi: 10.53560/PPASA(59-1)710.
[4] "Software Developer Jobs Will Increase Through 2026 | Dice.com Career Advice." https://www.dice.com/career-advice/software-developer-jobs-increase-2026 (accessed Oct. 02, 2023).
[5] M. S. Naveed, "Comparison of C++ and Java in Implementing Introductory Programming Algorithms," Quaid-E-Awam Univ. Res. J. Eng. Sci. Technol. Nawabshah., vol. 19, no. 1, pp. 95–103, Jun. 2021, doi: 10.52584/QRJ.1901.14.
[6] A. Moss, R. Schluntz, and P. A. Buhr, "C: Adding modern programming language features to C," Softw. Pract. Exp., vol. 48, no. 12, pp. 2111–2146, Dec. 2018, doi: 10.1002/SPE.2624.
[7] "View of C in CS1: Snags and Viable Solution." https://publications.muet.edu.pk/index.php/muetrj/article/view/91/45 (accessed Oct. 02, 2023).
[8] C. Sanderson and R. Curtin, "Armadillo: a template-based C++ library for linear algebra," J. Open Source Softw., vol. 1, no. 2, p. 26, Jun. 2016, doi: 10.21105/JOSS.00026.
[9] J. Gosling, B. Joy, G. Steele, G. Bracha, and A. Buckley, "The Java ® Language Specification Java SE 8 Edition," 1997.
[10] G. O'Regan, "The Innovation in Computing Companion," Innov. Comput. Companion, 2018, doi: 10.1007/978-3-030-02619-6.
[11] C. Severance, "Guido van Rossum: The early years of python," Computer (Long. Beach. Calif)., vol. 48, no. 2, pp. 7–9, Feb. 2015, doi: 10.1109/MC.2015.45.
[12] G. M. M. Bashir and A. S. M. L. Hoque, "An effective learning and teaching model for programming languages," J. Comput. Educ. 2016 34, vol. 3, no. 4, pp. 413–437, Jul. 2016, doi: 10.1007/S40692-016-0073-2.
[13] D. Rodriguez, I. Herraiz, and R. Harrison, "On software engineering repositories and their open problems," 2012 1st Int. Work. Realiz. AI Synerg. Softw. Eng. RAISE 2012 - Proc., pp. 52–56, 2012, doi: 10.1109/RAISE.2012.6227971.
[14] L. Ardito, R. Coppola, L. Barbato, and D. Verga, "A Tool-Based Perspective on Software Code Maintainability Metrics: A Systematic Literature Review," Sci. Program., vol. 2020, 2020, doi: 10.1155/2020/8840389.
[15] D. Knight, S. Torri, and T. Bhowmik, "A Preliminary Critical Review of the Impact of Three Popular Development Practices on Source Code Maintainability," Proc. - Int. Comput. Softw. Appl. Conf., vol. 2023-June, pp. 1633–1637, 2023, doi: 10.1109/COMPSAC57700.2023.00252.
[16] P. Quality, F. Sholichin, M. Adham, S. A. Halim, and M. Firdaus Bin Harun, "REVIEW OF IOS ARCHITECTURAL PATTERN FOR TESTABILITY," J. Theor. Appl. Inf. Technol., vol. 15, no. 15, 2019, Accessed: Oct. 02, 2023. [Online]. Available: www.jatit.org
[17] V. Garousi, M. Felderer, and F. N. Kılıçaslan, "A survey on software testability," Inf. Softw. Technol., vol. 108, pp. 35–64, Apr. 2019, doi: 10.1016/J.INFSOF.2018.12.003.
[18] M. S. Naveed, "Correlation Between GitHub Stars and Code Vulnerabilities," J. Comput. Biomed. Informatics, vol. 4, no. 01, pp. 141–151, Dec. 2022, doi:

10.56979/401/2022/111.

[19] "Analysis of Code Vulnerabilities in Repositories of GitHub and Rosettacode: A comparative Study | International Journal of Innovations in Science & Technology." https://journal.50sea.com/index.php/IJIST/article/view/289 (accessed Oct. 02, 2023).

[20] M. Scotto, A. Sillitti, and G. Succi, "AN EMPIRICAL ANALYSIS OF THE OPEN SOURCE DEVELOPMENT PROCESS BASED ON MINING OF SOURCE CODE REPOSITORIES," https://doi.org/10.1142/S0218194007003215, vol. 17, no. 2, pp. 231–247, Nov. 2011, doi: 10.1142/S0218194007003215.

[21] P. Bhattarai, M. Ghassemi, and T. Alhanai, "Open-source code repository attributes predict impact of computer science research," Proc. ACM/IEEE Jt. Conf. Digit. Libr., Jun. 2022, doi: 10.1145/3529372.3530927.

[22] D. L. Vu, I. Pashchenko, F. Massacci, H. Plate, and A. Sabetta, "Towards Using Source Code Repositories to Identify Software Supply Chain Attacks," Proc. ACM Conf. Comput. Commun. Secur., pp. 2093–2095, Oct. 2020, doi: 10.1145/3372297.3420015.

[23] R. Raducu, G. Esteban, F. J. R. Lera, and C. Fernández, "Collecting Vulnerable Source Code from Open-Source Repositories for Dataset Generation," Appl. Sci. 2020, Vol. 10, Page 1270, vol. 10, no. 4, p. 1270, Feb. 2020, doi: 10.3390/APP10041270.

[24] B. Norick, J. Krohn, E. Howard, B. Welna, and C. Izurieta, "Effects of the number of developers on code quality in open source software: A case study," ESEM 2010 - Proc. 2010 ACM-IEEE Int. Symp. Empir. Softw. Eng. Meas., 2010, doi: 10.1145/1852786.1852864.

[25] D. R. Wijendra, S. Lanka, and K. P. Hewagamage, "Analysis of Cognitive Complexity with Cyclomatic Complexity Metric of Software General Terms," Int. J. Comput. Appl., vol. 174, no. 19, pp. 975–8887, 2021.

[26] "The Impact of Language Syntax on the Complexity of Programs: A Case Study of Java and Python | International Journal of Innovations in Science & Technology." https://journal.50sea.com/index.php/IJIST/article/view/339 (accessed Oct. 02, 2023).

[27] M. Shepperd, "A critique of cyclomatic complexity as a software metric," Softw. Eng. J., vol. 3, no. 2, p. 30, 1988, doi: 10.1049/SEJ.1988.000310.1049/SEJ.1988.0003.

[28] A. Odeh, M. Odeh, N. Odeh, and H. Odeh, "Machine Learning Model for Measuring Cyclomatic Complexity of Source Code," 2023 Int. Conf. Intell. Comput. Commun. Netw. Serv. ICCNS 2023, pp. 149–153, 2023, doi: 10.1109/ICCNS58795.2023.10193630.

[29] M. S. naveed, "Measuring the Programming Complexity of C and C++ using Halstead Metrics:," Univ. Sindh J. Inf. Commun. Technol. , vol. 5, no. 4, pp. 158–165, 2021, Accessed: Oct. 02, 2023. [Online]. Available: https://sujo.usindh.edu.pk/index.php/USJICT/article/view/4073

[30] G. Yenduri and V. Naralasetti, "A Nonlinear Weight-Optimized Maintainability Index of Software Metrics by Grey Wolf Optimization," Int. J. Swarm Intell. Res., vol. 12, no. 2, pp. 1–21, Apr. 2021, doi: 10.4018/IJSIR.2021040101.