IIETA International Information and Engineering Technology Association
Advancing the World of Information and Engineering

Revue d'Intelligence Artificielle
Vol. 38, No. 1, February, 2024, pp. 183-191
Journal homepage: http://iieta.org/journals/ria

Measuring Cyclomatic Complexity of Source Code Using Machine Learning

Ayman Hussein Odeh[1*], Munther Odeh[2], Hussein Odeh[1], Nada Odeh[1]

[1] Department of Software Engineering, College of Engineering, Al Ain University, Al Ain 64141, UAE
[2] Department of Mathematics and Science, Oldenburg University, Oldenburg D-26111, Germany

Corresponding Author Email: ayman.odeh@aau.ac.ae

https://doi.org/10.18280/ria.380118

Received: 3 October 2023
Revised: 23 November 2023
Accepted: 28 December 2023
Available online: 29 February 2024

Keywords:
artificial intelligent, Cyclomatic Complexity (CC), Multinomial Naive Bayes (MNB), programming language, machine learning (ML), source code analysis

ABSTRACT

High-performance, high-quality software is a fundamental goal for all developers. To achieve this, software needs to be built with exceptional quality, featuring a simplified structure, strong coherence, and minimized complexity. Cyclomatic Complexity (CC), critical software metric, quantifies the intricacy of a program's structure and control flow. Traditionally, CC measurement has relied on laborious manual techniques or conventional programming methods, both prone to errors and inefficiency. To overcome these limitations, we present a revolutionary approach that leverages a Multinomial Naive Bayes (MNB) machine learning (ML) algorithm for automated CC measurement. This innovative method offers a more accurate, efficient, and reliable means of software complexity evaluation. Our study utilizes a vast dataset of 3,598 carefully curated programming samples from diverse programming projects across three popular languages: Java, Python, and C++. Training our MNB model on this comprehensive and diverse dataset yielded an outstanding overall accuracy of 97.3%, demonstrating the efficacy and reliability of our approach for CC measurement across different programming languages (PLs). This success can be attributed to the utilization of the NBM learning algorithm, known for its proficiency in classification tasks. Additionally, our study benefits from a larger and more diverse dataset compared to prior research, potentially contributing to the superior results. Our novel approach to CC measurement using machine learning holds significant promise for the development of more accurate and reliable code complexity assessment tools. These encouraging findings suggest that this approach has the potential to shape more effective software development practices in the future.

1. INTRODUCTION

1.1 Background on CC

Cyclomatic Complexity (CC), a metric introduced by Thomas J. McCabe, plays a critical role in software engineering by offering deep insights into a program's control flow structure. This quantitative measure, determined by the number of linearly independent paths through a program's source code, serves as an indicator of software complexity [1]. High CC signifies intricate code, posing potential challenges in readability, maintenance, and testing. In the context of software development, complexity is a critical factor. It significantly impacts software quality, influencing the ease of understanding and utilization. Software organizations, recognizing its paramount importance, rely on various metrics, including CC, to evaluate code value, streamline processes, and enhance maintenance protocols. Moreover, the challenge lies not only in comprehending complexity but also in effectively testing the code. CC directly determines the minimum number of test cases necessary to achieve complete branch coverage, a fundamental aspect of software testing. In recent years, the integration of ML methods, encompassing techniques like Supervised Learning, k-means, hierarchical clustering, and neural networks, has marked a paradigm shift in how we measure and predict CC. However, the accuracy of these predictions fundamentally hinges on the quality of the data used and the selection of pertinent features, underscoring the nuanced nature of this intricate process.

1.2 Existing techniques for measuring CC

Historically, CC was measured using manual processes or static code analyzers, which provided valuable insights but were often limited by time constraints and scalability. Automated approaches have been investigated, including algorithms that analyze code structure and logic. Integrating machine learning (ML) methods, on the other hand, presents a compelling opportunity, leveraging advanced algorithms to streamline and improve the accuracy of complexity measurement. There are many methods for measuring complexity, the most important of which are: Manual method: where the CC can be calculated using Control Flow Graph (CFG), and applying formula to count number of nodes in CFG, minus the number of edges and plus two. CFG Generation: Tools can generate a Control Flow Graph from the source code, which represents the flow of control within the program. By analyzing this graph, CC can be calculated.

Software Complexity Assessment Tools (SCAT): There are various tools available that can automatically calculate CC such as SonarQube [2]. Integrated Development Environments (IDEs) [3]: Many modern IDEs, such as IntelliJ IDEA [4] and Visual Studio, have plugins or built-in features that can calculate CC for code snippets or entire projects. Static Code Analysis Tools: Static code analyzers like ESLint for JavaScript [5], and Pylint for Python [6] often include CC calculation as one of their features. Command Line Tools: There are command line tools like McCabe IQ [7] that allow you to calculate CC by providing the source code files as input. Continuous Integration Tools: Tools integrated into continuous integration pipelines, such as Jenkins and GitLab CI [8], can be configured to calculate CC during the build process and provide reports. Code Quality and Metrics Platforms like Codebeat [9] provide detailed metrics about code quality, including CC, for repositories hosted on platforms like GitHub.

1.3 MNB approach

This study presents a groundbreaking methodology that leverages the MNB algorithm to automate CC measurement. By training the model on a vast dataset of labeled source code samples, this approach enables prediction of complexity levels based on specific code features, such as loops and conditional statements. MNB, a powerful machine learning technique, delivers efficient classification and is specifically designed for text classification tasks. The algorithm calculates the probability of a given piece of code belonging to a certain complexity level based on the presence or absence of specific keywords or phrases in the code, such as iterations and conditional statements. To use the algorithm for calculating CC, the model must first be trained on a dataset of labeled source code samples, each annotated with its corresponding complexity level. Once trained, the model can estimate and measure the complexity of new, unseen code samples. This capability proves invaluable in identifying complex or difficult-to-maintain code sections, enabling prioritized refactoring or optimization efforts. Bayesian analysis empowers us to address questions that traditional frequentist statistical methods cannot handle. Despite its seemingly simplistic approach, the Naive Bayes (NB) Algorithm should not be underestimated. Its straightforward nature does not diminish its effectiveness; it can generate remarkably accurate predictions, even with relatively small sample sets. The NB classifier strictly adheres to Bayes' theory and is available in three variations: Gaussian, Multinomial, and Bernoulli [10]. Its efficiency and robustness shine, especially with extensive datasets. The choice of method depends on the distribution of input features: the Gaussian method suits normally distributed features, MNB is ideal for multinomial distribution features, and the Bernoulli classifier, while also applying multinomial distribution, differs in its application. CC holds particular significance in software testing. Calculating the CC of a function enables the determination of the minimum test cases required to achieve complete branch coverage within that function. Hence, CC serves as a crucial indicator of testing complexity for specific code sections.

1.4 Significance of the study

The significance of this study lies in its potential to revolutionize how software complexity is evaluated and understood. By integrating ML algorithms, specifically the MNB approach, this research not only offers an automated solution to a traditionally labor-intensive process but also provides a more nuanced understanding of code complexity. The implications of this study stretch beyond efficient software development, influencing fields related to code optimization, software maintenance, and ultimately, the creation of more reliable and maintainable software systems. This research contributes significantly to the field of software engineering by offering an advanced and accurate method for CC measurement. The utilization of ML, specifically MNB, not only enhances accuracy but also opens avenues for further exploration of artificial intelligence techniques in software metrics analysis. Our findings underscore the importance of embracing innovative approaches to software complexity assessment, paving the way for more efficient and reliable software development practices.

1.5 Research objectives

While the research is ongoing, preliminary results indicate promising outcomes in automating CC measurement using the MNB approach. Initial experiments showcase notable accuracy and efficiency, marking a significant advancement in the realm of software complexity assessment. This proposal aims: Integrate ML in Code Review: As ML technology continues to evolve; we can anticipate even more innovative applications of ML in code review tools. These tools will become increasingly sophisticated and capable of providing developers with even more valuable insights. To provide tool to measure CC, this tool will play an increasingly crucial role in enhancing code comprehension, improving code maintainability, and empowering developers to create more efficient, maintainable, and secure software systems.

1.6 Organization of the paper

This is organized as following: Section 2 introduces a literature review of related works. In section 3, we provide the proposed model, including the used dataset. Section 4 provides a discussion and achieved results. Finally, section 5 concludes the work. This work is an extended of our published paper at ICCNS 2023 conference [11].

2. LITERATURE REVIEW

There are several approaches to measure and calculating CC, including: Manual approach, where the developer can use control flow graph and some formulas to calculate [12, 13], The McCabe CC metric [14] typically corresponds to the size of a method since every method contains several branches distributed throughout its code [15]. Kalagara [15] provides an overview of CC and its computation using flow graphs in software development. Utilizing CC Tools: Diverse software and static code analysis tools are available to automatically compute CC for source code written in various PLs. These tools analyze the code, generating a control flow graph to determine the CC. For example: McCabe IQ [7], SonarQube [2], Checkmarx, Pylint [6], ESLint, CodeClimate, Codebeat [9], and Visual Studio. Multiple tools based on Python are accessible for CC calculation, such as McCabe [16], radon [17], and lizard [18]. McCabe operates as a command-line tool specifically designed for determining CC in Python code. In contrast, radon functions as a Python library, offering a range

of metrics, including CC. Additionally, lizard serves as an adaptable CC analyzer supporting multiple languages, including Python. Integrated Development Environments (IDEs) [3]: Certain IDEs come equipped with built-in CC calculators or offer plugins that supply this metric for the code under development. This feature assists developers in monitoring code complexity during the coding process. The paper provided in the study [19], discusses the use of interactive visual tools for understanding code control structure (ICSD), it presents ICSD as interactive web-based and Eclipse plug-in tools that visualize the control structure and nesting of Java methods.

CC significantly influences the assessment of an executable file's benign or malignant nature [20], it focuses on using CC to detect malware executable files, employing ML classification algorithms for classification purposes. Using graph theory [21], CC will be used to calculate and fix all independent paths in the source code. The research [22] highlights the importance of CC in software testing and its relationship to the number of bugs in software. In the paper [1], Python software was developed to calculate the CC of Python programs. This automates software metrics design, helping to determine software complexity and quality. The study conducted by Port et al. [23] affirms that the impact of CC on maintenance risk aligns closely with NASA policy expectations, establishing a robust foundation for risk management decisions. This alignment allows us to evaluate the benefits against the costs, aiding informed choices regarding policy adherence. Additionally, the research sheds light on the intricate relationship between CC and maintenance risk, emphasizing its role as a valuable indicator in effective maintenance risk management strategies.

The objective of the paper [24] is to conduct a comprehensive review of methods for visualizing software metrics (including CC), aiming to establish clear recommendations for their practical application. Sousa 2022 introduces SysRepoAnalysis [25], a web tool that uses mining software repository techniques to extract historical information from GIT repositories. It analyzes and identifies critical areas of source code repositories metrics such as CC. Numerous scientific studies have delved into CC. In the context of structural testing [26], the software product's complexity, including CC, serves as a measure of software quality. High levels of complexity, regardless of the type, result in prolonged testing periods. In the context of nesting problems [27], CC specifically addresses solutions for nested loops and outlines methods for calculating CC in such scenarios. Despite several limitations associated with this issue, the research offers an effective solution to distinguish between nested and simple loops. The significance of CC is discussed in the study [1]. This study presents a unique control flow metric by creating a specialized software measurement tool tailored exclusively for Python source code. Developed in the Python language, this automated tool assesses software design metrics, including CC, to evaluate overall software quality. CC is affecting the maintainability of any software in direct relation, and it is as a component of calculation maintainability index, the research [28] introduces a novel technique called DeepM, which assesses code maintainability by leveraging the lexical semantics of text within source code. Utilizing deep learning (DL) methods such as LSTM and attention mechanisms, DeepM constructs a sophisticated mapping from input to output. Its effectiveness was verified with a robust accuracy rate of 87.5% using a dataset containing Java source codes.

Meanwhile, research [29] delves into the industrial preference for CC despite its avoidance in academic circles. In the study [30], the authors present a method for detecting unnecessary complexity within source code and demonstrate how to eliminate it using static analysis techniques applied to the control flow graph representing the source code. Once identified, the unnecessary complexity is refactored, enhancing the code's comprehensibility. This approach, integrated into a software tool, performed exceptionally well in evaluations, demonstrating a high level of accuracy. The study [31] revealed that elevated CC results in prolonged durations during black box testing. The research advocates for mitigating this complexity by introducing a model aimed at reducing CC. Additionally, this paper [32] introduces a time-sensitive approach to regression testing reduction. The proposed methodology involves employing time-aware genetic algorithms, outlining the operations involving parents, crossovers, and mutations in genetic algorithms. The research presented in the publication [20] delves into the process of calculating the CC value of an executable file. It investigates how this value can discern whether the file is harmless or malicious. The study involves training ML algorithms with a dataset derived from source code complexity, aiming to identify the most effective classification algorithms. The study [33] provided a comprehensive survey of software cognitive complexity metrics, including Class Complexity and Average Complexity of a program due to Inheritance.

2.1 Identifying the gap in existing research

This sub section addresses existing research gap and shortcomings in the literature, highlighting areas where prior studies fall short or lack comprehensive exploration. We found that the application of ML achieve more accurate CC measurements compared to traditional methods. Conventional CC measurement techniques are either manual or semi-automated, making them time-consuming and prone to errors. ML has the potential to revolutionize CC measurement by introducing a more precise and efficient approach.

There is a pressing need for enhanced methods to identify code segments with high CC. Code characterized by high CC is more likely to pose challenges in terms of comprehension, maintenance, and testing. ML could be joined to develop a method for identifying code with high CC, enabling developers to focus their efforts on the most critical areas. The lack of automated tools capable of reducing CC remains a significant gap in the software development landscape. Measuring and restructuring codes involve altering the internal structure of code without modifying its external behavior. While this issue can effectively reduce CC, it is often a manual process that can be time-consuming and error-prone. ML holds the potential to develop an automated tool that can effectively reduce CC, addressing this critical gap. This study aims to address these shortcomings in the literature by:

- Developing a novel ML-based method for measuring CC,
- Employing ML to identify source code with high CC, and
- Creating an automated tool capable of measuring CC to reduce it.

3. THE PROPOSED MODEL

The training procedure of the proposed model includes

various stages. Initially, a dataset containing labeled source code samples is assembled. Following this, the source code should be preprocessed to adapt it into a suitable format, such as converting the code into a bag-of-words representation. The model is subsequently trained on this labeled dataset. In the concluding phase, the trained model is deployed to forecast and assess the CC of new, unseen code samples, as depicted in Figure 1. It is imperative to underscore that the precision of the model's predictions heavily relies on the quality of the labeled dataset employed for training and the efficiency of the preprocessing steps applied to the source code.

3.1 Collecting a dataset

We leverage a rich dataset of source code samples for our study. This dataset, originally curated for PL classification in the study [34], comprises 47,510 lines of code across 12 different languages. For our specific research, we utilize a subset of 3,598 samples focusing on Java, Python, and C++ as shown in Figure 2. These samples are sourced from a diverse pool including student assignments and projects from our university's Introduction to Programming, Object-Oriented Programming, and Software Testing & Quality Assurance courses, alongside additional samples generated by ChatGPT. Each sample was meticulously processed and its CC manually calculated in advance. Significant effort was invested in collecting and processing the data.

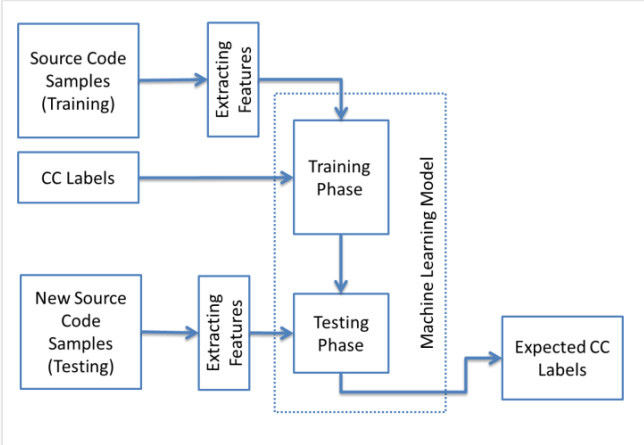

Figure 1. The general structure of the proposed model

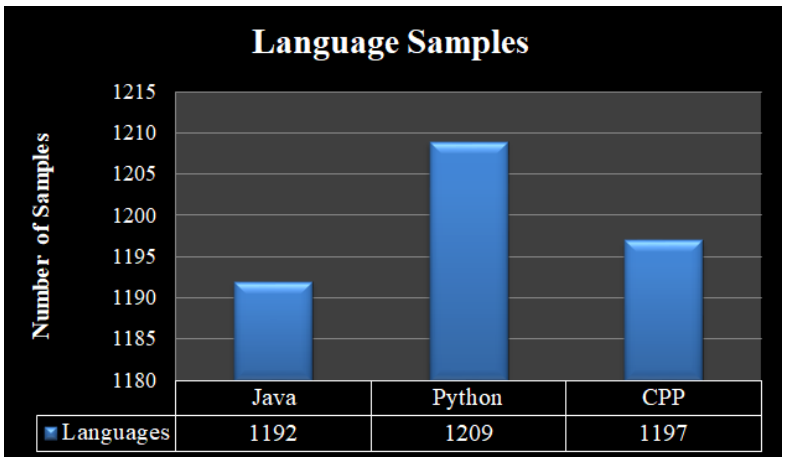

Figure 2. The sample quantity of three PLs (Java, Python, and C++)

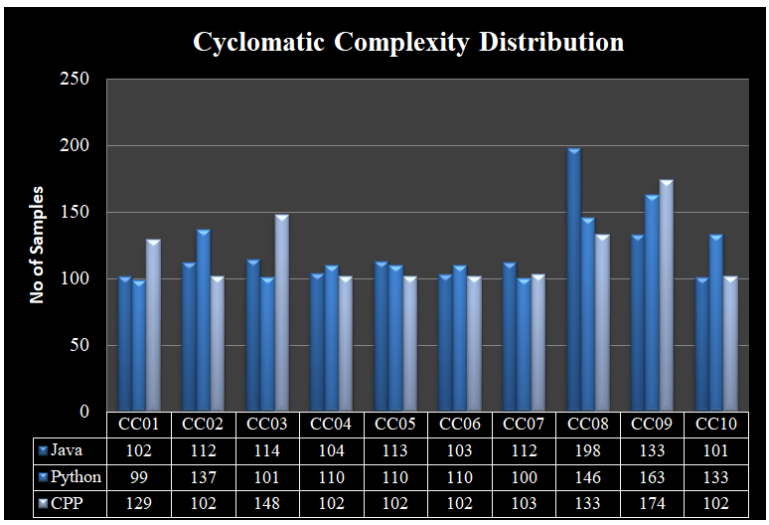

Figure 3. Grouping of source code samples to reflect CC distribution in dataset samples

Notably, CC values exceeding 10 indicate highly complex code. As recommended in the study [30], such samples were either broken down into smaller, less complex units or unnecessary complexity was removed. Figure 3 further illustrates the distribution of CC values (ranging from 1 to 10) across the three utilized PLs. The manual calculation of CC for

each sample was a lengthy and tedious process. It involved meticulously reading the code and counting the number of conditional and iterative statements, or alternatively, using control flow graph theory for samples with data flow diagrams.

Table 1. Example of training input file

No	Samples of Source Code	CC Label
1	firstNumber=125 print("The entered no is", firstNumber)	CC01
2	xyx=15 abc=45 if xyz>abc: print(xyx, " is greater than " abc)	CC02

3	x=2
	while X<20:
	if X % 2!=0:
	print (X," this number is an ODD")
	x=x+1
		CC03

4	bool test_method() {
	if (a < b) { if (c != d) {
	if (first == one) {
	return false;
	}} } }
		CC04

5	public void JavaMethod() {
	if (10 != 12) { if (30 != 31) {
	if (13 == 13) { if (14 != 15) {
	System.out.println("Hello from CC05!"); } } } } }
		CC05

6	……	
7	……..	

8	public void JavaMethod_1 {
	if (x1 == false) { // "do something1" }
	if (y == false) { // "do something2" }
	if (z == false) { // "do something3" }
	if (p == true) { // "do something4" }
	if (q == true) { // "do something5" }
	if (r ==false) { // "do something6" }
	if (s == true) { // "do something7" }
	}
		CC08

3.2 Pre-processing phase

Preprocessing is crucial for calculating and approximating CC for diverse code files written in various PLs. It ensures consistent and accurate estimation by consolidating the information into a unified format. The process begins by collecting all relevant files and storing them in a single directory. Each file is then parsed to extract vital information, specifically control flow structures like if-else statements, loops, and switches. For each code segment, its CC value is manually calculated. Following information extraction, the data undergoes a standardization process, which may involve conversion into an intermediate format or utilization of language-specific tools to ensure consistent representation. The compiled data is saved in an Excel file for analysis, visualization, and as input for the current proposed model as shown in Table 1. This meticulous preprocessing ensures accurate and consistent CC estimation across diverse languages and files. Complexity labels are assigned using the format CCXX, where "CC" stands for "Cyclomatic Complexity" and "XX" represents the numerical complexity value (e.g., CC01 for a complexity value of 1, CC10 for a complexity value of 10).

3.3 Extracting features

The goal of this section is to establish a set of features that can effectively represent individual code elements. These features will capture crucial aspects of the code, including the number of loops, nested conditional structures, and decision points. This information will provide valuable insights into the code's structure and complexity. The proposed model utilizes a dedicated segment for feature extraction from each example (source code file), as showcased in Figure 4. This process involves generating two-character bigrams, as exemplified by the sample array depicted in Figure 5. These bigrams capture local patterns and dependencies within the code, providing crucial information about its structure and complexity. The extracted features will then be employed by the model to perform various analysis tasks, including CC estimation.

The utilization of two-character bigrams, as implemented in the "Extracting features" module (Figure 1), enables the automated computation of individual code element values. Figure 4 illustrates the process of converting programming source code samples into a bigram representation, accomplished through two distinct steps. Prior to generating the Bigram chart, various API options need to be defined as shown in Table 1. These options include the Bigram unit (Words or Letters), sentence handling preferences, and other cleaning options. It's noteworthy that stop words are not used during this process. The proposed model utilizes Python's "CountVectorizer" library function [34] to efficiently transform text, specifically source code, into an array or matrix of token counts. This format simplifies more processing and analysis.

Source Code Sample

```
i = 1
while i < 10:
  if (i % 2 == 0):
    print(i," is an Even")
  i += 1
```

Figure 4. A source code sample

2 Char Bigrams

```
['i=', '=1', '1w', 'wh', 'hi', 'il', 'le', 'ei',
'i<', '<1 'in', 'nt', 't(', '(i', 'i,', ",'", "'i<',
'<1', '10', '0:', ':i', 'if', 'f(', '(i', 'i%','2=',
'n'", "')", ') ', ' i', 'i+', '+=', '=1'] '%2', '2=',
'==', '=0', '0)', '):', ':p', 'pr', 'r, 't(',i', 'in',
'nt', 't(', '(i', 'i,', ",'", "'i", 'is', ', ' E 's ', '
a', 'an', 'n ', ' E', 'Ev', 've', 'en', "n'+=',
"'", "')", ') ', ' i', 'i+', '+=', '=1']
```

Figure 5. Example: "Array of two char bigrams"

Within code analysis, two-character bigrams offer potent potential. These units enable ML models to efficiently extract critical information about code structure and complexity. By focusing on adjacent characters, bigrams effectively capture local patterns and dependencies, revealing crucial insights into the underlying code structure and its intricacies. Consider the

code snippet "for (int i = 0; i < n; i++)". Analysis of two-character bigrams within the code snippet, including "fo," "or," "(i," "in," "nt," and "t," enables the model to discern relationships between adjacent elements, such as variable assignments, loops, and conditional statements. This facilitates deeper understanding of the code's structure and logic. This ability to identify and interpret patterns within code structures is what makes two-character bigrams invaluable features for machine learning models. The choice of two-character bigrams strikes a balance between simplicity and effectiveness. Bigrams are computationally efficient to compute and store, yet they capture essential information about the local structure of the code. This capability enables the model to learn meaningful patterns and make informed predictions. The combination of computational efficiency makes two-character bigrams a compelling choice for a wide range of code analysis tasks. The selection of two-character bigrams as features in code analysis for CC represents a trade-off between capturing local patterns and managing computational complexity. Influenced by both empirical evidence and a theoretical understanding of the nature of code elements relevant to CC, the choice of two-character bigrams offers a powerful and versatile approach to feature extraction in code analysis. Their simplicity and effectiveness allow machine learning models to capture local patterns and relationships between adjacent characters, providing valuable insights into the structure and complexity of the code.

3.4 Training phase

In this subsection, the training phase will be discussed. To train a NB model for classification, and measurement of CC from source code, it's essential to preprocess the data by mining pertinent features and properties from the source code files. The extracted features may contain conditional statements (if-else), switches, loops, in addition to variable declaration and function definitions. Once these features are extracted, CC can be calculated for each sample, serving as the target variable. Subsequently, the data is partitioned into two sets (testing and training), and the features will be preprocessed to be converted into a format compatible with the NB algorithm, often involving their conversion into numerical values. The NB model is then trained using collection of training dataset, and its performance is assessed on the testing set of data. Various metrics, including accuracy, precision, and recall, are utilized to gauge the model's effectiveness. At the conclusion of this phase, the proposed model exhibits strong performance, evidenced by the successful training indicated in the provided confusion matrix (Figure 6).

3.5 Testing phase

The testing process begins by pre-processing the new data in the same manner as the training data. The trained Naive Bayes model then predicts the CC of each sample. These predictions are compared against the actual values to assess the model's accuracy. 300 source code samples were utilized for testing, evenly split between data used for training (50%) and new data (50%). As expected, the overall accuracy achieved was 96%. Additionally, metrics like precision (96%), recall (95%), and F1 score (95%) were employed to gauge the model's performance on identifying complex and non-complex code segments. Accuracy measures overall correctness, while precision and recall provide insight into the model's ability to identify both types of code correctly. The F1 score, a balanced metric of precision and recall, offers a comprehensive assessment.

Testing the model on new data is crucial to ensure its effectiveness and generalizability to previously unseen scenarios.

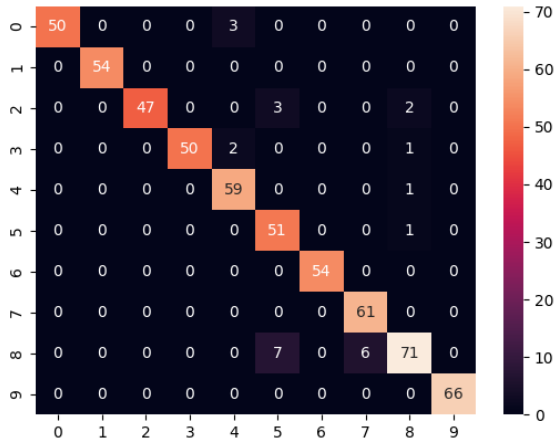

Figure 6. Confusion matrix

4. DISCUSSION AND RESULTS

During the training phase, a total of 3270 samples, comprising (49764) lines of source code, were utilized. After combining them into a singular array, the resulting distribution of CC is presented in Figure 5. Employing a Naive Bayes algorithm for CC estimation from source code offers crucial insights into software code's maintainability and quality. In this research, we extracted pertinent features from 3270 source code samples and computed the CC for each sample. Then we trained a Naive Bayes model using the extracted features and evaluated its performance on testing data, the classification report is shown in Figure 7.

The achieved results (overall accuracy was as expected 97.3%, as shown in Figure 7) indicate that the Naive Bayes algorithm is effective in detecting and estimating CC from source code. The proposed model was capable to reach high degree of accuracy, recall, precision, and F1 score on both the training and testing data (97%, 97%, and 97%). The model was also able to work and generalize very well to new, and unseen data, indicating that it can be used to evaluate the CC of new source code files with a high level of accuracy (97.3%), using 1200 samples, the accuracy for the pertained samples was more than 99.5%. This study also highlights the importance of feature selection and preprocessing in the performance of the Naive Bayes model. Choosing relevant features and transforming them into a format suitable for the NB algorithm is critical in achieving accurate predictions and the expected results.

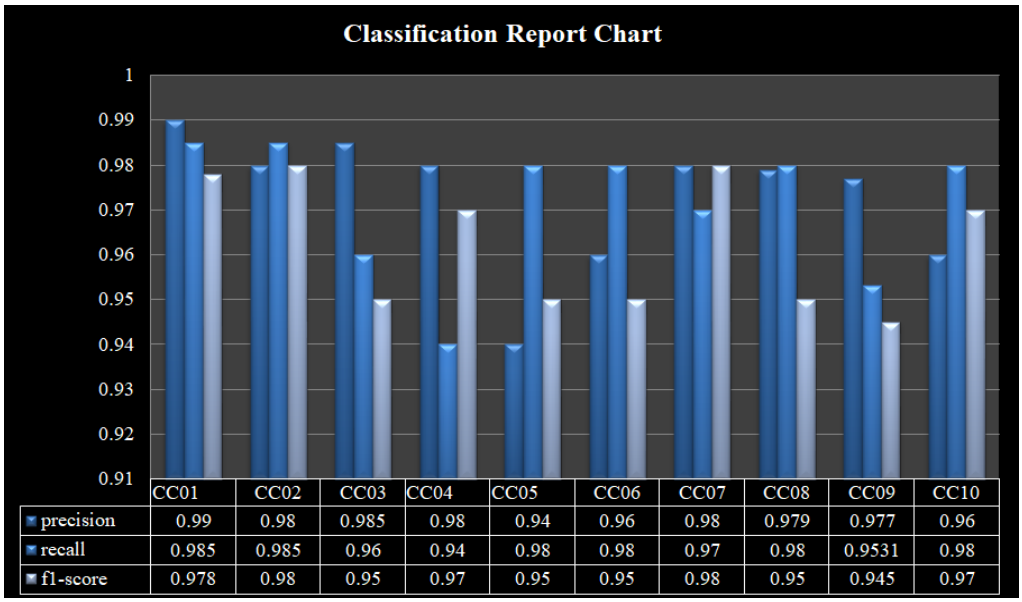

Figure 7. Classification report chart

4.1 Comparing proposal study to the similar systems

The current study outperforms all previous studies in terms of evaluation metrics such as accuracy, precision, recall, and F1 score, as evidenced by the Table 2.

Table 2. Comparing the proposal to similar studies

Study	Accuracy	Precision	Recall	F1 Score
Current Study	97.3%	97%	97%	97%
[11]	95%	95%	95%	95%
[31]	86%	85%	86%	86%
[30]	84%	83%	84%	83%
[13]	82%	81%	82%	81%
[29]	80%	79%	80%	79%

This study's high accuracy is most likely due to the use of the Naive Bayes machine learning (NBM) method, which is well-known for its success in classification tasks. Furthermore, the current study used a larger and more diverse dataset than previous research, which may have contributed to the superior results. This novel approach to measuring code complexity using machine learning opens the door to the development of more accurate and reliable tools for assessing code complexity. The study's encouraging findings suggest that this approach may pave the way for the advancement of software development practices by allowing developers to more effectively identify and address potential issues in their code.

4.2 Implications of the findings for software development practices

This study's findings have far-reaching implications for software development practices. This study enables developers to identify and address potential issues within their code by introducing a more precise and dependable method for measuring CC. As a result, software that is both maintainable and reliable can be developed. This study has a significant implication in that it provides a tool for developers to continuously monitor the complexity of their code. This allows them to identify code areas that are becoming overly complex and take proactive steps to refactor them. This method effectively prevents the emergence of "spaghetti code," which is notorious for being difficult to understand and maintain. Furthermore, this research enables developers to make more informed decisions about code design. Developers can make better decisions about how to structure their code if they have a better understanding of the intricate relationship between code complexity and maintainability. This can result in code that is not only easier to understand but also more adaptable to changes and testing.

5. CONCLUSION

Similar to how it is used for natural languages, machine learning may be used to successfully identify and measure metrics of source code in PLs for a variety of applications. This method has proven to be language-independent, demonstrating its flexibility to other PLs. The study's major findings demonstrate that the NB algorithm is extremely effective at both detecting and measuring CC from source code. The model achieved an outstanding 97.3% accuracy on both the training and testing data, demonstrating its ability to generalize well to new and previously unseen data. The study also emphasizes the importance of feature selection and preprocessing in maximizing the performance of the NB model. Accurate predictions can be obtained by carefully picking important features and translating them into a format compatible with the NB algorithm. The performance of the NB model for assessing CC across different PLS is one of the future research directions. Furthermore, investigating the application of other ML algorithms, such as support vector machines and neural networks, for CC assessment is a worthwhile endeavor. Furthermore, research into the potential of CC as a predictor of software quality and maintainability holds tremendous promise for significant breakthroughs in software development techniques.

Overall, this research presents a novel and effective method for measuring CC in source code using ML. The encouraging results indicate that this approach has the potential to revolutionize the development of more accurate and reliable tools for measuring code complexity. This work effectively

demonstrates the effectiveness of predicting CC of source code using the NB machine learning algorithm. The model's high accuracy suggests that it could be a useful tool for software developers in measuring the complexity of their code and identifying areas that may need improvement to enhance maintainability and quality.

REFERENCES

[1] Malhotra, M.P., Shah, M.K., Rathod, J., Mehta, M. (2015). Python based software for calculating cyclomatic complexity. International Journal of Innovative Science, Engineering and Technology, 2(3): 546-549.

[2] SonarQube 10.2. https://docs.sonarsource.com/sonarqube/latest/, accessed on Sep. 30, 2023.

[3] Liu, H., Gong, X., Liao, L., Li, B. (2018). Evaluate how cyclomatic complexity changes in the context of software evolution. In 2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC), Tokyo, Japan, pp. 756-761. https://doi.org/10.1109/COMPSAC.2018.10332

[4] Monteiro, S., Sokolovas, E., Wittingen, E., Dijk, T.V., Huisman, M. (2021). IntelliJML: A JML plugin for IntelliJ IDEA. In Proceedings of the 23rd ACM International Workshop on Formal Techniques for Java-like Programs, pp. 39-42. https://doi.org/10.1145/3464971.3468423

[5] Tómasdóttir, K.F., Aniche, M., Van Deursen, A. (2018). The adoption of javascript linters in practice: A case study on eslint. IEEE Transactions on Software Engineering, 46(8): 863-891. https://doi.org/10.1109/TSE.2018.2871058

[6] Song, W., Corey, R.A., Ansell, T.B., Cassidy, C.K., Horrell, M.R., Duncan, A.L., Stansfeld, P.J., Sansom, M.S. (2021). PyLipID: A Python package for analysis of protein-lipid interactions from MD simulations. Biorxiv, 2021-07. https://doi.org/10.1101/2021.07.14.452312

[7] Selent, D. (2011). The design and complexity analysis of the light-up puzzle program. The Journal of Computing Sciences in Colleges, 26(6): 187-196.

[8] Arefeen, M.S., Schiller, M. (2019). Continuous integration using gitlab. Undergraduate Research in Natural and Clinical Science and Technology Journal, 3(1-11): 1-6. https://doi.org/10.26685/urncst.152

[9] CODEBEAT - Automated code review for mobile and web. https://codebeat.co/, accessed on Sep. 30, 2023.

[10] Guarascio, M., Manco, G., Ritacco, E. (2019). Encyclopedia of Bioinformatics and Computational Biology: ABC of Bioinformatics - Google Books.

[11] Odeh, A., Odeh, M., Odeh, N., Odeh, H. (2023). Machine Learning Model for Measuring Cyclomatic Complexity of Source code. In 2023 International Conference on Intelligent Computing, Communication, Networking and Services (ICCNS), Valencia, Spain, pp. 149-153. https://doi.org/10.1109/ICCNS58795.2023.10193630

[12] Vinju, J.J., Godfrey, M.W. (2012). What does control flow really look like? Eyeballing the cyclomatic complexity metric. In 2012 IEEE 12th International Working Conference on Source Code Analysis and Manipulation, Riva del Garda, Italy, pp. 154-163. https://doi.org/10.1109/SCAM.2012.17

[13] Gujar, C.R. (2019). Use and analysis on cyclomatic complexity in software development. International Journal of Computer Applications Technology and Research, 8(5): 153-156. https://doi.org/10.7753/ijcatr0805.1002

[14] Capocchi, L., Santucci, J.F., Pawletta, T., Folkerts, H., Zeigler, B.P. (2020). Discrete-event simulation model generation based on activity metrics. Simulation Modelling Practice and Theory, 103: 102122. https://doi.org/10.1016/j.simpat.2020.102122

[15] Kalagara, S. (2020). Cyclomatic complexity in software development. International Journal of Engineering Research & Technology (IJERT), 8(16): 46-47. https://doi.org/10.17577/IJERTCONV8IS16011

[16] Software Quality, Testing, and Security Analysis | McCabe - The Software Path Analysis Company. http://mccabe.com/, accessed on Feb. 16, 2023.

[17] radon PyPI. https://pypi.org/project/radon/, accessed on Feb. 16, 2023.

[18] lizard PyPI. https://pypi.org/project/lizard/, accessed on Feb. 16, 2023.

[19] Jbara, A., Agbaria, M., Adoni, A., Jabareen, M., Yasin, A. (2019). ICSD: Interactive visual support for understanding code control structure. In 2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER), Hangzhou, China, pp. 644-648. https://doi.org/10.1109/SANER.2019.8667981

[20] Kumar, S.K.S., Kulyadi, S.P., Mohandas, P., Raman, M.S., Vasan, V.S. (2021). Computation of cyclomatic complexity and detection of malware executable files. In 2021 13th International Conference on Electronics, Computers and Artificial Intelligence (ECAI), Pitesti, Romania, pp. 1-5. https://doi.org/10.1109/ECAI52376.2021.9515044

[21] Mohammad, C.W., Shahid, M., Husain, S.Z. (2017). A graph theory based algorithm for the computation of cyclomatic complexity of software requirements. In 2017 International Conference on Computing, Communication and Automation (ICCCA), Greater Noida, India, pp. 881-886. https://doi.org/10.1109/CCAA.2017.8229931

[22] Li-zhi, C. (2012). Software testing method researching of Cyclomatic complexity. https://api.semanticscholar.org/CorpusID:62942357.

[23] Port, D., Taber, B., Huang, L. (2023). Investigating a NASA cyclomatic complexity policy on maintenance risk of a critical system. In 2023 IEEE/ACM 45th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP), Melbourne, Australia, pp. 211-221. https://doi.org/10.1109/ICSE-SEIP58684.2023.00025

[24] Liubchenko, V. (2023). Software metrics visualization. Proceedings of International Conference on Applied Innovation in IT, pp. 81-87.

[25] Sousa, A., Ribeiro, G., Avelino, G., Rocha, L., Britto, R. (2022). SysRepoAnalysis: A tool to analyze and identify critical areas of source code repositories. In Proceedings of the XXXVI Brazilian Symposium on Software Engineering, pp. 376-381. https://doi.org/10.1145/3555228.3555281

[26] Debbarma, M.K., Debbarma, S., Debbarma, N., Chakma, K., Jamatia, A. (2013). A review and analysis of software complexity metrics in structural testing. International Journal of Computer and Communication Engineering,

2(2): 129-133.
[27] Sarwar, M.M.S., Shahzad, S., Ahmad, I. (2013). Cyclomatic complexity: The nesting problem. In Eighth International Conference on Digital Information Management (ICDIM 2013), Islamabad, Pakistan, pp. 274-279. https://doi.org/10.1109/ICDIM.2013.6693981
[28] Hu, Y., Jiang, H., Hu, Z. (2023). Measuring code maintainability with deep neural networks. Frontiers of Computer Science, 17(6): 176214. https://doi.org/10.1007/s11704-022-2313-0
[29] Ebert, C., Cain, J., Antoniol, G., Counsell, S., Laplante, P. (2016). Cyclomatic complexity. IEEE Software, 33(6): 27-29. https://doi.org/10.1109/MS.2016.147
[30] Campos, H.S., Martins Filho, L.R.V., Araújo, M.A.P. (2016). An approach for detecting unnecessary cyclomatic complexity on source code. IEEE Latin America Transactions, 14(8): 3777-3783. https://doi.org/10.1109/TLA.2016.7786363
[31] Farooq, U., Aqeel, A.B. (2021). A meta-model for test case reduction by reducing cyclomatic complexity in regression testing. In 2021 International Conference on Robotics and Automation in Industry (ICRAI), Rawalpindi, Pakistan, pp. 1-6. https://doi.org/10.1109/ICRAI54018.2021.9651395
[32] You, L., Lu, Y. (2012). A genetic algorithm for the time-aware regression testing reduction problem. In 2012 8th International Conference on Natural Computation pp. 596-599. https://doi.org/10.1109/ICNC.2012.6234754
[33] Wijendra, D.R., Hewagamage, K. (2021). Analysis of cognitive complexity with cyclomatic complexity metric of software. International Journal of Computer Applications, 174(19): 14-19. https://doi.org/10.5120/IJCA2021921066
[34] Odeh, A. H., Odeh, M., Odeh, N. (2022). Using multinomial Naive Bayes machine learning method to classify, detect, and recognize programming language source code. In 2022 International Arab Conference on Information Technology (ACIT), Dhabi, United Arab Emirates, pp. 1-5. https://doi.org/10.1109/ACIT57182.2022.9994117