Jeong-Jin Kang
Edward J. Rothwell
Yang Hao
Sang-Hyun Lee

Advanced and Applied Convergence Letters AACL 22

Advanced and Applied Convergence & Adavanced Culture Technology

11th International Symposium, ISAAC 2023
in Conjunction with ICACT 2023, ICKAI 2023

November 16 - 18, 2023, Jeju, Korea
Revised Selected Papers

IIBC The Institute of Internet, Broadcasting and Communication
IPACT The International Promotion Agency of Culture Technology
지식의 숲 FOREST OF KNOWLEDGE

Measuring Software Complexity of Other Similar Structured Softwares through Learning the Characteristics of a Single High Level Language

*Chansol Park, **Jinmo Yang, ***JiHoon Kong, ****R. Young Chul Kim

*Dept. of Software and Communication Engineering, Hongik University, Republic of Korea
**Dept. of Physics, Korea University, Republic of Korea
***Toonsquare
****Dept. of Software and Communication Engineering, Hongik University, Republic of Korea
*c2193102@g.hongik.ac.kr, **yjmd2222@gmail.com,
***john.tooning@toonsquare.co, ****bob@hongik.ac.kr

Abstract

As various platforms and devices continue to be developed, the number of programming languages is also increasing. Software developed using these new technologies requires quality assessment. However, to measure the quality of new types of software using traditional software engineering methods, new parsers and quality measurement tools for new programming languages must be developed. This approach presents a problem as it demands significant costs and effort. In this study, we propose a complexity measurement method for software written in similar languages through the learning of characteristics of a specific high-level language, thereby measuring complexity, which is one of the quality indicators in software. We collect datasets through an existing rule-based visualization tool. We train a large language model using the collected dataset, allowing the trained model to measure quality indicators for languages with similar structures.

Keywords*: Software Engineering, Software Quality, Software Complexity, Artificial Intelligence, Large Language Models*

1. Introduction

As technology advances, various devices and platforms are continuously being developed. Consequently, the variety of new programming languages suitable for these platforms and devices is also on the rise. Software designed for these new platforms and devices requires quality assessment like conventional software. To measure the quality of software using traditional software engineering methods, a parser for the programming language must be implemented to perform parsing. Subsequently, rules for quality measurement must be set to assess the quality. The challenge is that implementing new parsers and analysis tools requires substantial effort and costs. To address this issue, we propose a method for measuring the complexity of software written in structurally similar but different languages through the learning of characteristics of a high-level language. We collect quality datasets through already implemented quality measurement tools. Using the gathered quality datasets, we train a large language model. This trained model can measure the quality of software written in other structurally similar programming languages. This approach allows the measurement of software quality without the overhead of implementing parsers and measurement tools.

2. Related works

2.1 Code Visualization-Based Software Data Collection Toolchain

J.J. Kang et al. (Eds.): ISAAC 2023, AACL 22, 2023

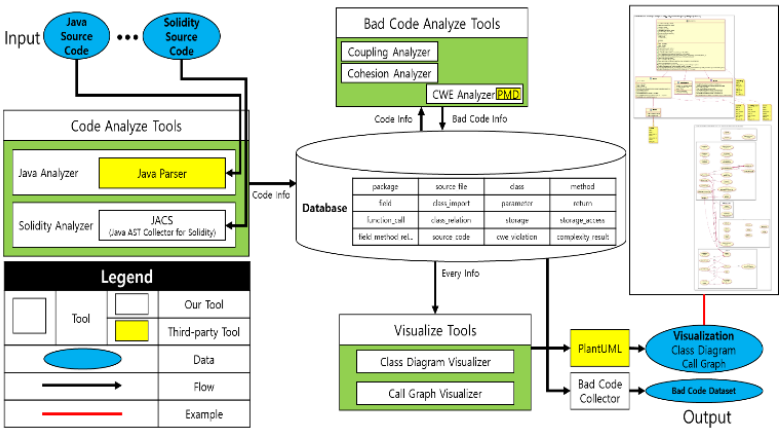

Figure 1. Code Visualization-Based Software Data Collection Toolchain

Figure 1 shows the structure of the code visualization-based software data collection toolchain[1]. The toolchain analyzes the code, detects code complexity and vulnerabilities, visualizes them, and creates a dataset. Bad Code Collector stores complexity, code, and vulnerability information as datasets for complex code. Complex code is more likely to contain vulnerabilities. Therefore, only complex code is stored to reduce the impact of false positives detected by the tool.

2.2 Identifying bad codes through supervised learning with bad code patterns

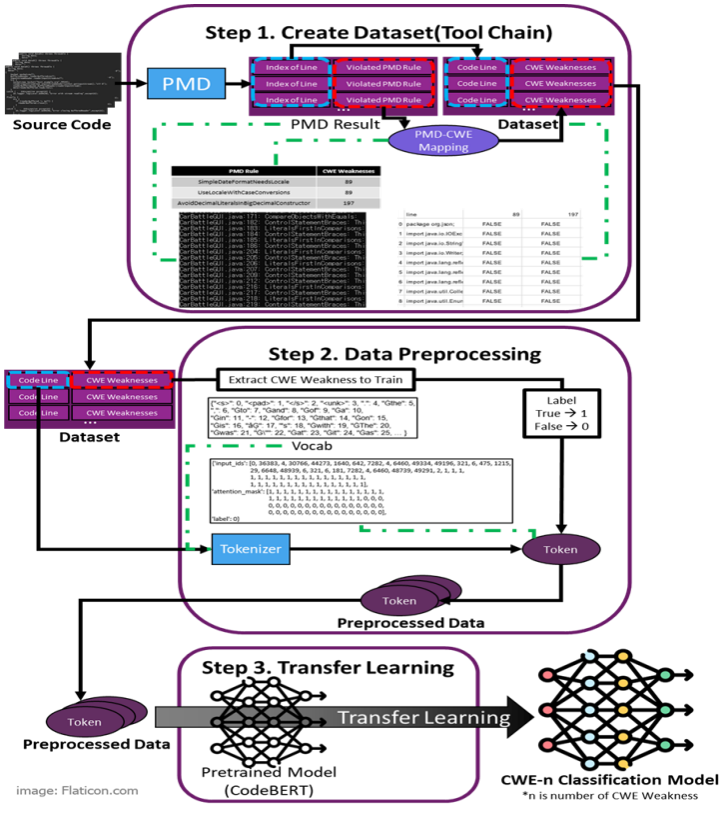

Figure 2. Sequence for Learning CWE Items in CodeBERT model

Figure 2 is a flowchart illustrating how the CodeBERT model[2] learns the patterns of Bad Code to identify Common Weakness Enumeration (CWE)[3] entries[4]. The training process of CWE entries for the CodeBERT model consists of three stages: Create Dataset, Data Preprocessing, and Transfer Learning. In the Create Dataset stage, the Programming Mistake Detector (PMD) tool [5] is used to collect labeled data for CWE vulnerabilities. Vulnerabilities detected by the PMD tool are converted into CWE vulnerabilities through a mapping table and labeled accordingly. In the Data Preprocessing stage, code lines are tokenized, and the detection status of a single CWE entry to be learned by the model is labeled. The transfer-learned CodeBERT model identifies CWE vulnerabilities in code lines.

3. High level language complexity measurement large language model

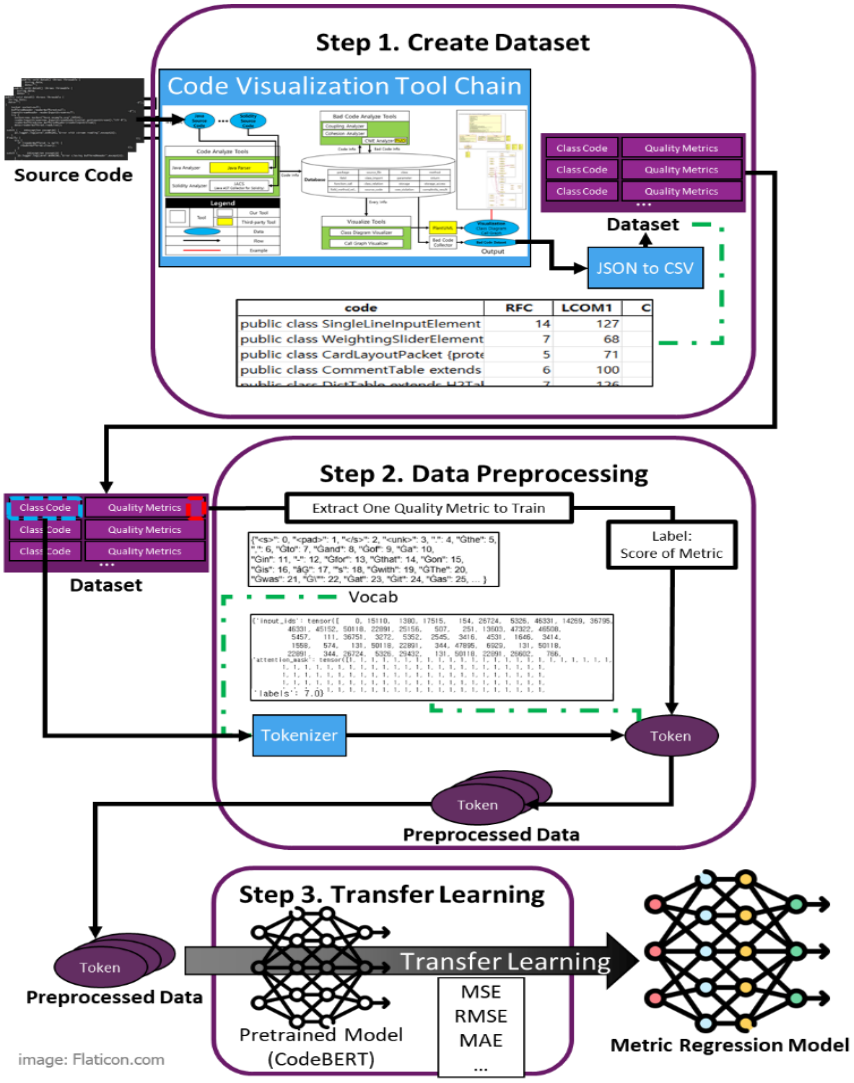

Figure 3. Sequence for Learning Complexity in CodeBERT model

To train the complexity of high level languages, CodeBERT is used as a large language model. Figure 3 represents the flowchart of training high-level language complexity to a large language model. The stages for training are divided into Create Dataset, Data Preprocessing, and Transfer Learning.

In the Create Dataset stage, JSON-format data collected using code visualization-based software data

collection tool is transformed into a dataset consisting of source code and complexity. During this process, irrelevant annotations, packages, import information, and comments are removed.

In the Data Preprocessing stage, the source codes of classes are tokenized, and the complexity for training is labeled. In the transfer learning stage, the CodeBERT model is trained using mean squared error as the loss function to solve the regression problem. The CodeBERT model utilizes a vocabulary dictionary that maps tokens during the tokenization process of source code. This vocabulary dictionary is used across different programming languages. Therefore, the model can be applied to software written in structurally similar languages, even if the language used for model training is different.

4. Application

As an application, the complexity of the JAVA language is measured and trained in CodeBERT. For the application of cohesion, LCOM[6] is trained in the model. LCOM represents the number of method pairs within a class that do not use common attributes. For the application of coupling, RFC[7] is trained in the model. RFC represents the number of methods that can potentially be executed by responding to messages received by objects of a class. The trained model is then applied to examples in object-oriented programming languages like C++, C#, Python, and Kotlin, in addition to JAVA, to measure complexity.

Table 1. Comparison table of complexity measured by rule-based tools for code written in JAVA and complexity measured by CodeBERT model for code written in JAVA and similar object-oriented languages(C++, C#, Python, Kotlin)

<table>
<tr><td>Language</td><td colspan="2">JAVA</td></tr>
<tr><td>Code</td><td colspan="2">

```
public class Team {
    character mainLeader;
    ArrayList<character> team;
    private int effectiveLevel;
    private int money;
    public int getMoney() { return money; }
    public void addMoney(int addition) { money += addition; }
    public int getEffectiveLevel() { return updateEffectiveLevel(); }
    private int updateEffectiveLevel() {
        int el = team.get(0).getLevel();
        for (int i = 1; i < team.size(); i++) {
        }
        return el;
    }
    public Team(character leader) {
        mainLeader = leader;
        team = new ArrayList<character>();
        effectiveLevel = 0;
        money = 100;
    }
    public void addTeamMember(character newMember) { team.add(newMember); }
    public void removeTeamMember(character member) { team.remove(member); }
}
```

</td></tr>
<tr><td>Complexity</td><td>Measured by Rule-based</td><td>Measured by CodeBERT</td></tr>
<tr><td>LCOM</td><td>72</td><td>239.79277</td></tr>
<tr><td>RFC</td><td>8</td><td>7.4962177</td></tr>
<tr><td>Language</td><td>C++</td><td>C#</td></tr>
<tr><td>Code</td><td>

```
class Team {
    public:
        character mainLeader;
        list<character> team;
    private:
        int effectiveLevel;
        int money;
    public
        int getMoney() { return money; }
        void addMoney(int addition) { money += addition; }
        int getEffectiveLevel() { return updateEffectiveLevel(); }
        Team(character leader) {
            mainLeader = leader;
            team = {-1};
            effectiveLevel = 0;
            money = 100;
        }
        void addTeamMember(character newMember) { team.push_back(newMember); }
        void removeTeamMember(character member) { v.erase(remove(v.begin(), v.end(), member), v.end()); }
    private
        int updateEffectiveLevel() {
            int el = team[0].getLevel();
            for (size_t i = 1; i < team.size(); i++) {
            }
            return el;
        }
}
```

</td><td>

```
class Team{
    character mainLeader;
    List<character> team;
    private int effectiveLevel;
    private int money;
    public int GetMoney(){ return money; }
    public void addMoney(int addition){ money += addition; }
    public int getEffectiveLevel(){ return updateEffectiveLevel(); }
    private int updateEffectiveLevel(){
        int el = team[0].Level;
        for (int i = 1; i < team.Count; i++){
        }
        return el;
    }
    public Team(character leader){
        mainLeader = leader;
        team = new List<Character>();
        effectiveLevel = 0;
        money = 100;
    }
    public void addTeamMember(character newMember){ team.Add(newMember); }
    public void removeTeamMember(character member){ team.remove(member); }
}
```

</td></tr>
<tr><td>LCOM</td><td>239.79277</td><td>239.79277</td></tr>
<tr><td>RFC</td><td>6.3979707</td><td>7.5715055</td></tr>
<tr><td>Language</td><td>Python</td><td>Kotlin</td></tr>
<tr><td>Code</td><td>

```
class Team:
    def __init__(self, leader):
        self.mainLeader = leader
        self.team = []
        self.effectiveLevel = 0
        self.money = 100
    def get_money(self): return self.money
    def add_money(self, addition): self.money += addition
    def getEffectiveLevel(self): return self.updateEffectiveLevel()
    def addTeamMember(self, newMember): self.team.append(newMember)
    def removeTeamMember(self, member): self.team.remove(member)
    def update_effective_level(self):
        el = self.team[0].level
        for i in range(1, len(self.team)):
            pass
        return el
```

</td><td>

```
class Team(leader: character) {
    var mainLeader: character = leader
    var team: MutableList<character> = mutableListOf()
    private var effectiveLevel: Int = 0
    private var money: Int = 100
    fun getMoney(): Int { return money }
    fun addMoney(addition: Int) { money += addition }
    fun getEffectiveLevel(): Int { return updateEffectiveLevel() }
    fun addTeamMember(newMember: character) { team.add(newMember) }
    fun removeTeamMember(member: character) { team.remove(member) }
    private fun updateEffectiveLevel(): Int {
        var el = team[0].level
        for (i in 1 until team.size) {
        }
        return el
    }
}
```

</td></tr>
<tr><td>LCOM</td><td>239.79277</td><td>239.79277</td></tr>
<tr><td>RFC</td><td>10.507485</td><td>5.8400884</td></tr>
</table>

Table 1 presents the complexity of the classes in the JAVA code and those in the code translated to C++, C#, Python, and Kotlin measured by the CodeBERT model. In the case of LCOM complexity, the model produced exactly the same values for all the code. This implies that the model did not learn LCOM complexity accurately. For RFC complexity, the correct answer measured by rule-based code visualization tools is 8. The results obtained through the CodeBERT model were the most accurate for C# code, with approximately 7.57. Following that, the results for JAVA, C++, and Kotlin were similar to the correct answer. The measurement for Python code was the least accurate, with a result of 10.507485.

The CodeBERT model trained on data labeled with complexity for JAVA classes can provide a reasonable approximation of complexity. However, for metrics like LCOM, which involve complex rules, the model may not learn well. Additionally, when measuring code written in other similarly structured languages, it is observed that the measurement accuracy is higher for programming languages with structures more similar to JAVA.

4. Conclusion and future works

In this paper, we have proposed a method for measuring the complexity of high-level languages using a large language model. The large language model trained on the complexity of high-level languages can measure the complexity of software written in structurally similar languages. This allows the measurement of complexity for software written in new types of languages without implementing parsers and quality measurement tools. As future work, we plan to perform multi-regression using large language models to measure a broader range of complexity and quality metrics with a single trained model.

Acknowledgement

This research was supported by Culture, Sports and Tourism R&D Program through the Korea Creative Content Agency(KOCCA) grant funded by the Ministry of Culture, Sports and Tourism(MCST) in 2023(Project Name: Development of AI-based user interactive multi-modal interactive storytelling 3D scene authoring technology, Project Number: RS-2023-00227917, Contribution Rate: 100%).

References

[1] C.S. Park, W.S. Jang, and R.Y.C. Kim, “Tool Chain Mechanism with Identifying and Collecting High Quality Data for Learning Bad Code based on Code Visualization,” 2023 Conference, Korean Institute of Smart Media, Vol. 12, No. 1, pp. 52-53, 2023.

[2] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, et al. "Codebert: A pre-trained model for programming and natural languages," arXiv preprint arXiv:2002.08155, 2020.

[3] Common Weakness Enumeration, http://cwe.mitre.org.

[4] C.S. Park, S.Y. Moon, R.Y.C. Kim, “Detecting Common Weakness Enumeration(CWE) Based on the Transfer Learning of CodeBERT Model,” *KIPS Transactions on Software and DATA Engineering*, Vol. 12, No. 10, pp. 431-436, 2023.

[5] PMD, https://pmd.github.io/.

[6] S.R. Chidamber and C.F. Kemerer, “Towards a metrics suite for object oriented design,” SIGPLAN, Vol. 26, No. 11, pp. 197-211, 1991.

[7] B. Henderson-Sellers, “Object-oriented metrics: Measures of complexity,” Prentice-Hall, pp.142-147, 1996.