Code Clone Detection Analysis

22343 - Mustafa Ata Onbas*
26748 - Mert Kılıcaslan[α]

Sabanci University Department of Engineering, Computer Science & Engineering

ABSTRACT

This paper addresses the enduring issue of managing code clones in the field of software engineering, an issue that often results in code redundancy, increased maintenance efforts, and a potential decrease in software quality. We present a comprehensive, user-friendly framework that incorporates three distinct code clone detection tools: SimpleCC, Duplicate Code Detection, and our tool, myCC. SimpleCC and Duplicate Code Detection have been borrowed from existing literature, while myCC is a unique contribution aimed at enhancing detection capabilities. These tools synergistically work to alleviate the difficulties associated with identifying and managing code clone pairs.

The utility of our framework was validated using the bigCloneBench dataset from Microsoft's CodeXGLUE repository, a dataset containing 9,126 lines of code transformed into separate Java files to emulate a real-world scenario. This diverse dataset facilitated an extensive evaluation of the correctness and similarity of detected clone pairs by different tools, thus establishing the framework's effectiveness.

Our framework is designed for analysis of code clone detections, drawing inspiration from the work of Juergens, Deissenboeck, Hummel, and Wagner (2009). A distinguishing feature of our framework is its ability to allow users to switch between different tools and results, thereby offering a more tailored and flexible approach to code clone detection. Our work aims to provide an unbiased evaluation of the performance and reliability of different code clone detection algorithms and a user-friendly solution to code clone management.

KEYWORDS: Code Clone Detection; SimpleCC, Duplicate Code Detection, MyCC, bigCloneBench.

1 – INTRODUCTION

In the field of software engineering, the persistent challenge of managing code clones – identical or similar code fragments scattered throughout a software system – constitutes an ongoing challenge of utmost importance. The existence of these clones often induces issues such as code redundancy, heightened maintenance effort, and potential deterioration in software quality (Roy, Cordy, and Koschke, 2009). Given these concerns, we have developed a comprehensive, user-friendly framework that incorporates three distinct code clone detection tools. Two of these tools, SimpleCC (a Type2 detection tool), and Duplicate Code Detection (a Type1 tool), are borrowed from existing literature. Additionally, we have contributed our own solution to this mix, a unique tool as myCC, which also operates as a Type2 detection tool. The creation of myCC signifies our effort to enhance the detection capabilities in the existing tools. Along with these, the bigCloneBench dataset deployed to test the effectiveness and capability of the integrated tools and the overall framework structure.

These tools, under the part of our comprehensive framework, work synergistically to mitigate the difficulties in detecting and managing code clone pairs. SimpleCC, as its name suggests, employs a

*aonbas@sabanciuniv.edu
[α]mkilicaslan@sabanciuniv.edu
https://github.com/SU-CS442-22SP/Team02_CodeCloneDetectionAnalyser

straightforward but effective strategy. It involves a four-class program that extracts a list of tokens, implements a normalization rule, and detects clones using a longest common substring extraction approach (Kamiya, Kusumoto, and Inoue, 2002). Duplicate Code Detection, on the other hand, employs a robust string diff check and natural language processing for Type1 code clone pairs and overall similarity result. The tool we developed, myCC, builds on the principles of SimpleCC but aims to be less complex, making it a more user-friendly and efficient tool for certain tasks, the myCC tool uses a simplified version of SimpleCC's method for code clone detection, where it streamlines the process of extracting tokens and implementing normalization rules.

Incorporating insights from Juergens, Deissenboeck, Hummel, and Wagner (2009), who highlighted the impact of code clones on system complexity and maintenance cost, our framework is designed to significantly reduce these complications and costs by enabling users to see the results from their chosen tool, enhancing usability and efficiency. This ability to toggle between tools and results is an outstanding feature that distinguishes the framework, allowing users to adopt a more customized and flexible approach to see the code clone pairs and detection results between the types for a specified dataset.

2 – DATASET

The dataset used for our project was obtained from Microsoft's CodeXGLUE repository, which hosts a variety of datasets for various code-related tasks. Specifically, we employed the data.jsonl file from the Clone-detection-BigCloneBench directory for our code clone detection algorithm analysis. The CodeXGLUE repository is publicly accessible, and it provides a diverse set of datasets for machine learning tasks related to code, making it an ideal choice for our project (source).

The data.jsonl file contains lines of dummy code, including classes and methods, mimicking the structures typically seen in real-world programming scenarios. As our analysis is focused on Java code clone detection, we converted each line of this dataset into separate Java files. This was done to emulate the real-world scenario of analyzing entire files more accurately for code clones, as well as to align the format of the dataset with the requirements of the clone detection tools used in our study.

In total, there are 9,126 lines in the data.jsonl file. Accordingly, this process resulted in a corpus of 9,126 individual Java files that were subsequently analyzed by various code clone detection tools. This rich dataset allows for a comprehensive evaluation and comparison of these tools in terms of their correctness and similarity in detected clone pairs. By examining such many code samples, our project aims to provide a thorough and unbiased evaluation of the performance and reliability of different code clone detection algorithms.

3 - METHODOLOGY

3.1 – Duplicate Code Detection Tool

The Duplicate Code Detection tool is a key component of our code clone detection framework. This command-line utility, written in python, accepts either a directory or a list of files and assesses the degree of similarity between them. Initially developed for the DAT265 - Software Evolution Project, the tool serves to guide developers in their refactoring efforts, targeting the reduction of code duplication and the enhancement of software architecture.

The tool's main functionality relies on the gensim Python library, a sophisticated library renowned for its document similarity analysis capabilities. The tool applies gensim to assess the similarity between user-

supplied source code files. It supports a extensive range of programming languages, which are C, C++, JAVA, Python, and C#. While executing its tasks, the tool leverages several Python packages including Natural Language Toolkit (nltk), gensim, astor, and punkt. The nltk package and the punkt tokenizer facilitate text processing within the code files, while the astor package provides the means to manipulate and analyze Python's Abstract Syntax Tree (AST).

In the adaptation of the Duplicate Code Detection tool for our framework, we incorporated a robust string check using the Python difflib library. This library provides functionalities for comparing sequences, including lines of text, in a detailed and efficient manner. The integration of difflib considerably improves the tool's capacity to identify Type 1 clone pairs and display them in detailed manner, which are essentially identical code blocks, unaffected by variations in whitespace, layout, or comments.

The operation of the Duplicate Code Detection tool begins with processing the source code files. It tokenizes the text and creates a model representing the code. Utilizing the gensim library, the tool then compares these models to calculate a similarity score. Notably, before the calculation starts, the tool filters out and skips the similarity scores below a user-provided threshold, thereby maintaining focus on the most substantial code duplications. Subsequently, it calls the difflib-powered strong string difference checker to yield a detailed similarity report. The accuracy of the similarity calculations is noted to increase with a larger number of source code files, making this tool especially suitable for bigger projects.

The tool generates an output in the form of a comprehensive report, listing pairs of files along with their corresponding similarity scores. This facilitates easy interpretation and supports targeted refactoring efforts. Our framework, by integrating Natural Language Processing (NLP) and string comparison using difflib, amplifies its robustness in managing Type 1 code clone pairs, thereby enhancing the reliability of our clone detection strategy.

3.2 – SimpleCC

We selected SimpleCC, a type-2 clone detector, as one of the main tools for this project. Developed by Takashi Ishio, SimpleCC is a basic and lightweight tool designed for clone detection. It primarily works by tokenizing the source code into smaller components and comparing these tokens to identify similar sequences that suggest code clones.

3.2.1 – Modification of CloneDetectionMain

We modified the CloneDetectionMain class from the original SimpleCC tool. This class serves as the main entry point for the clone detection process.The readFile method reads a file at the specified path, tokenizes the file's contents, and returns an ArrayList of CodeToken objects. It leverages the Java8Lexer class from the ANTLR library to convert the source code into tokens.

The detectClones method accepts two ArrayLists of CodeTokens and a threshold. This method is responsible for finding potential clone pairs in the two provided token lists. For every pair of tokens in the two lists, if the tokens are from different files and identical, it will look ahead for any more matching tokens. If the length of the matching sequence is greater than or equal to the provided threshold, it will report this as a potential clone pair. The reportClone method prints information about the detected clone pair, including the path, starting line number, and character position of both the beginning and end of the clone sequences in each file.

3.2.2 – SimpleCCExample Class

We created a helper class, SimpleCCExample, to facilitate the process of reading the dataset, creating the Java files, and running SimpleCC on all pairs of files.

The main method first reads the dataset from the data.jsonl file and converts each line into a separate Java file. Then it loops over all pairs of these files, calling SimpleCC for each pair. The output from SimpleCC is captured and, if a clone pair is detected, the information about this pair is written to a file named clones.txt.

The SimpleCC tool is executed through a ProcessBuilder, which runs the CloneDetectionMain class with the two file paths as arguments. The output of SimpleCC is read and processed to extract the clone information. If the process takes longer than 2 minutes, it is terminated. If any errors occur during the process execution, these are also reported. This class greatly simplifies the process of applying SimpleCC to a large dataset by automating the tasks of data preparation and tool execution. In the context of the SimpleCCExample class, the ProcessBuilder embodies the essence of the Command design pattern. The ProcessBuilder object represents a command to be executed in the system's environment. The command to run SimpleCC with specific arguments (the paths of the pair of files) is encapsulated within the ProcessBuilder object, effectively decoupling the initialization of the command from its execution.

The ProcessBuilder class prepares the command to be run in a separate process, and when the start() method is called on the ProcessBuilder object, it executes the command and returns a Process object that represents the running process. In this manner, the command's execution can be controlled and its results can be retrieved.

The use of ProcessBuilder showcases the flexibility and extensibility of the Command design pattern. The command's parameters can be easily modified (by changing the file paths), new commands can be added without changing the existing code (by creating new ProcessBuilder objects with different commands), and the execution of commands can be controlled in a uniform manner (through the Process object).
This implementation of the Command design pattern with ProcessBuilder is integral to the robustness and adaptability of our SimpleCCExample class. It allows us to run SimpleCC on all possible pairs of files in a systematic and controlled manner, while also making it easy to adapt the code for different commands or datasets in the future.

3.3 – MyCC

The myCC tool is a Type-2 code clone detection system. Code clones, particularly Type-2 clones, are identical program structures that differ in variable identifiers, literals, types, layout, and comments. Detecting and managing these clones is an important task in maintaining software quality, as clones can often be the source of bugs and inconsistencies.

The tool functions by performing a deep analysis of Java source code files, breaking them down into Abstract Syntax Trees (ASTs) to represent their structure and contents. This representation allows for a detailed comparison between code segments at the statement level.

3.3.1 – Source Code Parsing

The first step in the process involves parsing the source code files to generate their respective ASTs. The parse method performs this action, reading the source code from the file and utilizing the Eclipse JDT Core's ASTParser to generate the AST. If the source code doesn't contain a class, a dummy class is added for the successful generation of the AST.

3.3.2 – File Path Gathering

To analyze multiple files, myCC gathers all the Java file paths from a given directory using the getFilePaths method. This method walks through the directory and its subdirectories, filtering only the paths that point to Java files.

3.3.3 – Statement Normalization

A crucial step for Type-2 clone detection is normalizing statements. myCC does this in the normalizeStatement method, which replaces all variable identifiers with a generic "var" and changes all number and string literals to generic placeholders. This ensures that differences in variable names and literals don't prevent detection of structurally identical code.

3.3.4 – Code Clone Detection

MyCC then performs the clone detection process. It iterates over pairs of files, breaking down each into statements using the getStatementsFromCompilationUnit method. Each statement from one file is compared to every statement in the other files using the isSimilar method, which employs ASTMatcher's safeSubtreeMatch function to check structural similarity between statements.

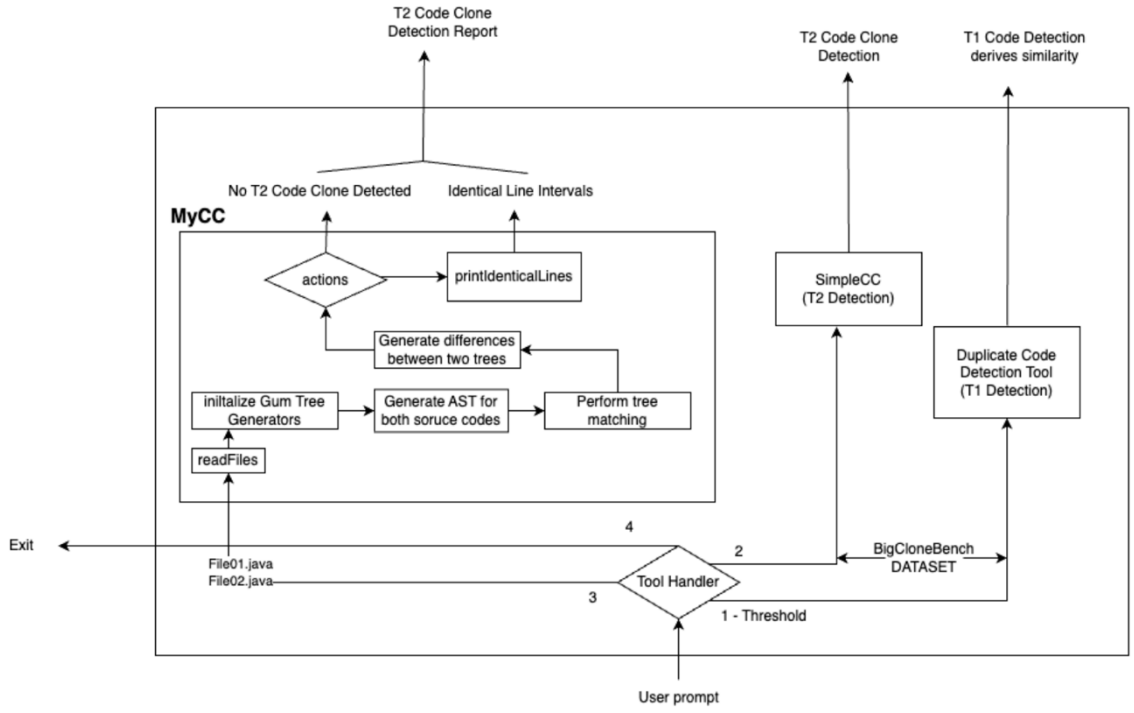
3.3.5 – Clone Reporting

If a clone pair is found (two statements that are similar), myCC records the clone pair in a text file. This is accomplished in the writeCloneData method, which writes file paths and the cloned statements to the output file.

3.3.6 – Execution

The execution of the algorithm begins in the main method. Here, file paths are collected from the provided dataset directory, and findClones is called to initiate the clone detection process.

3.4 – General Functionality of the Abstract Framework



Our implemented abstract framework is command-line-based, offering a user-centric approach that takes inputs directly from the user. It leverages the functionality of a tool handler that allows the user to navigate through four distinct capabilities of the framework, each accessible via entering a corresponding numerical input.

The first functionality, invoked by the input '1', is the Duplicate Detection Tool. This tool employs the bigCloneBench dataset and initiates a comprehensive comparison between all files in the dataset. It generates a similarity score for each pair of files, thereby highlighting potential code clones and their degree of similarity. This information is relayed back to the user in the output, providing a straightforward and accessible way to identify potential duplicate code.

The second functionality, accessible via input '2', triggers the SimpleCC tool. SimpleCC specifically targets the detection of Type 2 code clones. Upon activation, this tool scans the dataset, identifies code clone pairs and presents the user with a list of these pairs in the output. This aids in the identification and management of more subtle, semantic code clones that can often be overlooked.

The third functionality, initiated by input '3', activates our novel tool, MyCC. MyCC builds upon the principles of SimpleCC but introduces an abstract tree-based comparison technique. By utilizing this method, MyCC offers a less complex, but more user-friendly and efficient approach to code clone detection. Upon execution, MyCC compares the dataset files, identifies code clone pairs, and returns the results in the output, making it an ideal tool for detecting and managing code clones in a more efficient manner.
The final command, invoked by the input '4', offers users the capability to exit the tool, thus providing a means to end the clone detection process when desired.

Overall, the general functionality of our abstract framework is geared towards enhancing usability and efficiency in managing code clones, offering a streamlined and straightforward approach for users to engage with multiple code clone detection tools.

4 – RESULTS & DISCUSSION

We have derived a similarity score by comparing these files with the help of Duplicate Code Detection algorithm:

File A	File B	Similarity
file_2.java	file_3.java	70.87
file_1.java	file_14.java	60.27
file_22.java	file_10.java	26.82
file_25.java	file_23.java	35.27
file_25.java	file_24.java	50.36
file_25.java	file_7.java	28.39
file_15.java	file_16.java	71.83
file_34.java	file_33.java	85.11
file_34.java	file_36.java	36.83
file_34.java	file_35.java	39.81
file_34.java	file_32.java	48.83
file_49.java	file_47.java	25.05
file_14.java	file_1.java	60.27
file_3.java	file_2.java	70.87
file_33.java	file_34.java	85.11
file_33.java	file_36.java	43.12
file_33.java	file_35.java	46.18

Fig.1 Part of the similarity scores

We have derived the clones detected by the SimpleCC and MyCC which are type-2 detection tool, here below is a part or template of how the pair detection is manipulated:

```
<pair>
/Users/mataonbas/IdeaProjects/SimpleCC/trialDataset/file_2.java,12,12,23,13
/Users/mataonbas/IdeaProjects/SimpleCC/trialDataset/file_3.java,1,12,12,13
<pair>

Object   run(){
try{
        MessageDigest   digest    =MessageDigest .getInstance("SHA");
        digest.update(buf.toString().getBytes());
        byte[] data = digest.digest();
        serialNum = new BASE64Encoder().encode(data);
        return serialNum;
}

catch(NoSuchAlgorithmException exp){
        BootSecurityManager.securityLogger.log( Level.SEVERE   ,exp.getMessage (),exp);
        Return buf.toString();
```

```
        }
}
```

Our results provide insight into the functioning and relative performance of the three code clone detection tools employed in our framework. One of the prominent outcomes of our research is represented in Figure 1, which depicts the similarity scores derived from the bigCloneBench dataset for some selected files.

Both SimpleCC and MyCC tools utilized the same template for output. An exemplary output is structured as follows: the output begins with a <pair> tag followed by the paths of two Java files in comparison, their associated lines of code, and finally a snippet of code. This standardized template of output allows users to easily navigate through the results and identify the code clone pairs.

Our findings also point out the limitations in our newly developed tool, MyCC. Despite its user-friendly approach and abstract tree-based comparison technique, it does not perform as accurately as SimpleCC in detecting Type-2 clones. This suggests that there is room for improvement in myCC's design and highlights the importance of further development and fine-tuning.

When comparing the outputs of SimpleCC and the Duplicate Code Detection Tool, we found a high degree of overlap in their results, suggesting both tools are reliable and efficient in detecting clones. Despite the similarity, SimpleCC exhibited a slight edge due to its ability to provide additional information about the code detections in terms of different classes, methods, and variables. However, the Duplicate Code Detection Tool was also successful in determining files that contained clones, marking it as a reliable option for clone detection. Moreover, we found a strong correlation between the high similarity scores from the Duplicate Code Detection tool and the clone pairs detected by SimpleCC. This finding validates the accuracy of the similarity scores produced by the Duplicate Code Detection Tool, and confirms its utility in assisting in the identification of potential code clones.

In summary, our results showcase the utility and strengths of each tool, and also highlight areas for further refinement and development. Our findings contribute valuable insights into the relative effectiveness of different clone detection tools and underscore the significance of our framework in providing a user-friendly and efficient approach to code clone management.

5 – CONCLUSION

This study presented an effective, comprehensive, and user-friendly framework for managing the persistent challenge of code clones in the software engineering field. By incorporating three distinct tools - SimpleCC, Duplicate Code Detection, and our tool, myCC - we delivered a flexible, tailored approach for detecting and managing code clone pairs. Each tool offered unique strengths, providing an encompassing solution to tackle different types of code clones.

The Duplicate Detection Tool, making use of the bigCloneBench dataset, provided a robust mechanism to assess the similarity between files. SimpleCC efficiently detected Type-2 clones, while our new tool, myCC, introduced a less complex approach, with scope for further refinement and improvement. Both SimpleCC and MyCC used a standardized output template, enhancing the user experience by maintaining a consistent result presentation.

Our evaluation, using a substantial corpus of Java files, illustrated the effectiveness and reliability of each tool. Despite differences in the clone detection approach, both SimpleCC and Duplicate Code Detection Tool yielded largely overlapping results, suggesting their dependable performance. Furthermore, the Duplicate Code Detection Tool's similarity scores strongly correlated with SimpleCC's detection of clone pairs, reaffirming its accuracy.

Although our results were promising, the performance gap between myCC and SimpleCC revealed opportunities for future work. This endeavor could revolve around refining myCC's design and enhancing its detection capabilities, particularly for Type-2 clones.

6 – REFERENCES

1. Roy, C. K., Cordy, J. R., & Koschke, R. (2009). Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. Science of Computer Programming, 74(7), 470-495.
2. Juergens, E., Deissenboeck, F., Hummel, B., & Wagner, S. (2009). Do code clones matter?. In Proceedings of the 31st International Conference on Software Engineering (ICSE '09) (pp. 485-495). IEEE.
3. Kamiya, T., Kusumoto, S., & Inoue, K. (2002). CCFinder: a multilinguistic token-based code clone detection system for large scale source code. IEEE Transactions on Software Engineering, 28(7), 654-670.
4. Dişli, H., & TOSUN, A. (2020). Code Clone Detection with Convolutional Neural Networks. *Bilişim Teknolojileri Dergisi*, *13*(1), 1-12.