UNIVERSITY OF CALIFORNIA,
IRVINE

Experimentation with Tokenization Process of SourcererCC

THESIS

submitted in partial satisfaction of the requirements
for the degree of

MASTER OF SCIENCE

in Software Engineering

by

Paridhi Sirohi

Thesis Committee:
Professor Cristina Lopes, Chair
Assistant Professor Iftekar Ahmed
Associate Professor James Jones

2022

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

I would like to thank Professor Cristina Lopes, my advisor, for allowing me to work on this project. The work on this thesis will not have been possible without her guidance and support. I would also like to thank Professor Iftekhar Ahmed and Professor Jim Jones for being on this committee.

Furthermore, I would like to thank my family for the support and encouragement I have always received. I also want to thank my close friends, Gaurang and Divyangna, for inspiring me and helping me put my best foot forward.

Finally, I would like to thank everyone I met at UC Irvine. Mansi, Sakshi, Iris, Samyak, Pooja, Rakib, and Ella, thank you for all the support and memories I have made here.

ABSTRACT OF THE THESIS

Experimentation with Tokenization Process of SourcererCC

By

Paridhi Sirohi

Master of Science in Software Engineering

University of California, Irvine, 2022

Professor Cristina Lopes, Chair

This thesis presents the experimentation of parameters affecting the tokenization process of an existing code clone detection tool, SourcererCC. The SourcererCC is a token-based clone detector that targets three clone types and exploits an index to achieve scalability to large inter-project repositories using a standard workstation. We experiment with the three parameters affecting tokenization: (1) threshold, (2) stop words, and (3) use of sub-tokens. I will evaluate these three parameters' performance with the original SourcererCC. I will be using the metrics, precision, and recall for the evaluation. I create a web interface for the SourcererCC and use that to conduct preliminary experiments for the parameters and find the best results with an 80 percent threshold.

The experiments conducted for the evaluation include (1) Original SourcererCC, (2) SourcererCC with sub-tokens enabled, (3) SourcererCC CC with stop words, and (4) SourcererCC with sub-tokens and stop words. The experiments are evaluated with a recall study using the BigCloneEval tool and manual verification of the precision of the experiments. For the manual verification, 150 samples are selected from each experiment, using four judges to remove any biases. I further analyze the results achieved and the scope of future work for this thesis.

Chapter 1

Introduction

Tokenization is the process of dividing a stream of characters into larger units called *tokens* [4]. Tokenization has applications across multiple domains, namely data mining, machine learning, information retrieval, linguistics, compilers and statistics. In recent years, we have seen exponential growth in the need for tokenization in computer learning models [28].

In the software engineering domain, tokenization is the first step for detecting code duplication. Reusing code is a common practice in software development. This can include copying and using it as it is or modifying the code based on the change in the use case [22]. These snippets of reused code create *clones*. Clones are similar pieces of code found within the project or across different projects. My thesis is about studying the effects that different tokenization approaches have on the effectiveness of code clone detection.

1.1 Background

Code clone detection focuses on identifying and labeling clones. There are mainly four types of clones [23]:

- Type 1: Identical code statements with differences in layout, white spaces, and comments
- Type-2: In addition to conditions of Type 1, the code statements may use different names for identifiers and literal values
- Type-3: Syntactically similar code fragments that differ at the statement level. The fragments have statements added, modified, and/or removed with respect to each other, in addition to Type-1 and Type-2 clone differences
- Type-4: Syntactically dissimilar code fragments that implement the same functionality

The code clone detection tool used in this thesis is SourcererCC [24]. SourcererCC is a token-based detection tool that identifies clone types 1, 2, and 3. It uses an index to provide scalability for detecting clones in large repositories. SourcererCC consists of two steps: (1) tokenization of the source code, and (2) detection of clones based on token comparisons.

In code clone detection software the tokens are generated from the source code. Source code is written to be interpreted by human programmers as well as computers [31]. This results in code having the syntax and grammar of a programming language for the computer and informal and formal conventions followed to make the code readable by the programmers. This can include various formatting techniques, comments, naming identifiers, etc.

Two important parameters affect the process of tokenization for code clone detection tools, namely: (1) generating subtokens (or not) and (2) using stop words (or not).

- **Subtokens.** In most programming languages, identifiers follow specific conventions, which include camel casing [16] and underscore. A subtoken is a semantically meaningful portion of the main token that can be obtained by knowing these conventions. For example: findDifference is a token that contains two subtokens (find and difference); total_marks is another token that contains two subtokens (total and marks); etc.

Using *subtokens* as a parameter for tokenization can affect the effectiveness of clone detection.

- **Stop words.** Stop words are high-frequency tokens that increase the percentage of similarity but do not hold the same importance as the non-frequent words. In the case of code clone detection, the programming language's keywords can be used as the stop words.

The tokenization and stop word removal is part of the preprocessing of data as depicted in Figure 1.1.

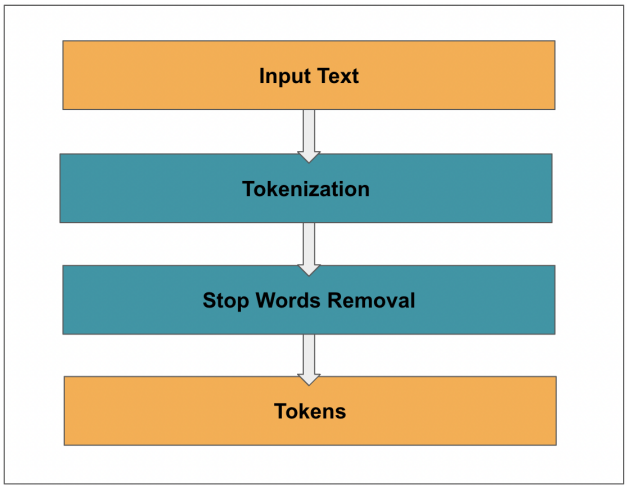

Figure 1.1: Preprocessing of Data

SourcererCC creates partial inverted indexes on the files with the tokens generated in the first step. The second step uses a clone-matching algorithm to find pairs of clones. During this process, the common tokens between the two code snippets are measured in percentage. The algorithm uses a configurable parameter, *threshold*, as the minimum percentage required for two code snippets to be considered clones. As such, another important parameter for code clone analysis of the threshold of similarity.

1.2 Motivation

Code clone detection tools are essential for the following reasons [22]:

- Plagiarism of code: The tool helps identify the code similarity and that is helpful in plagiarism. It can also help in law proceedings in case of copyright infringement.
- Increasing the libraries used by various languages: By analyzing large datasets of code, one can identify the code duplicated the most and if it is a common implementation, it can be incorporated into a library for future use.
- Bug reporting: Reusability of code is a common principle in object-oriented programming. In the case of code duplications, a bug reported at one of the instances is more likely to be a bug at the clones too. A quick code clone detection process can help in eliminating a common bug occurring due to duplication easily.
- Detection of malicious implementation: After detecting code clones, if one of the code fragments is a malicious implementation, one can classify all the clones as malicious implementations.

Current code clone detection tools present two issues. Firstly, achieving the best effectiveness of code clone detection for every new dataset requires experimentation with different parameters of the configurations [29]. However, there isn't much study and analysis on how the parameters in the configurations affect the evaluation metrics. Specifically for SourcererCC, an investigation of the effect of subtokens and stop words has not been done yet.

Secondly, current code clone detection tools are time-consuming to set up and use. Most of these tools require local code setup and the use of the command line [7], which limits both the amount of experimentation and the kinds of users that can drive the tools.

My thesis builds on these two issues. First, I present a study on the effect of subtokens and stop words in SourcererCC. Then I explain the design and implementation of a web interface that facilitates experimentation with configuration parameters.

1.3 Thesis Statement

The issues mentioned above in Section 1.2 spark the requirement for studying the effectiveness that different tokenization approaches have on the results of SourcererCC. This thesis will answer the following research questions:

- How does the use of subtokens and stop words affect the effectiveness of SourcererCC?
- Is there a compound effect in using subtokens and stop words in the effectiveness of SourcererCC?

In order to answer these questions, I designed experiments comparing the use of SourcererCC with and without subtokens and stop words, and I present the results here. To facilitate my own experiments, I designed and implemented a Web interface that facilitates these experiments by automating many setup and configuration steps required for running SourcererCC. The contributions of my work are as follows: (1) my experiments show that, while the use of subtokens and stop words, in separate, deteriorates the effectiveness of SourcererCC, their combined use results is an improvement in recall; (2) the Web interface proved to be an invaluable tool for my own experiments, saving me many hours of manual setup; this tool is publicly available and can be used by others.

Chapter 2

Literature Survey

This section covers the literature review. The literature review consists of three sections: (1) Previous work done in code clone detection tools with an emphasis on the SourcererCC, (2) literature review of tokenization across different domains, and (3) tokenization work done in the field of Natural Language Processing (NLP) models.

2.1 Previous Work

Clones are called by different names in the literature. Baxter et al. [1] termed it " identical to another code segment." Meanwhile, Komondoor [10] and Ducasse et al. [2] have used "duplicate code." Krinke et al. [17] have used the term "similar code." For simplicity, I will be using code clones in this thesis. Code clone detection tools have been a mature research year for the last two decades. Ratten et al. [21] presented a survey of over seventy code clone detection software in 2011. Roy and Chordy [23] have evaluated the benchmarks for the last decade in 2018. Shobha et al. [3] have conducted a systematic review of the code clone detection software based on the techniques used. The techniques used include text-based,

token-based, tree-based, graph-based, and hybrid.

This thesis intersects with token-based code clone detection, and I will focus on these moving forward. We will be covering the various token-based code clone detections present today. Baker et al. [1] introduced Dup tool for detecting clones by line-based string matching approach. It focuses on the change of variable names by an editor. The limitation of this approach is that it focuses on the main structure, and if the code is rewritten, it cannot catch duplicate code. Kamiya et al. [8] focused on large-scale code, including COBOL, C, and Java, for effectively extracting clones. Li et al. [13] introduced CP-Miner, which uses data mining methods for detecting clones in large-scale code. Deissenboeck et al. developed the CloneDetective tool for token-based clone detection technique by tokenization, which uses the suffix tree method to compare and detect clones. Kawaguchi et al. [9] present the Shinobi tool, which is implemented as a client-server structural design suffix array index method used for token sequence. Murakami et al. [18] developed an efficient FRISC tool using a suffix array algorithm with higher precision and recall. Sajnani et al. [24] presented SourcererCC token-based detection tool, which uses an optimized partial index and demonstrated scalability by IJaDataset open-source Java systems and compared it with CCFinder, Deckard, iClones, and NiCad.

In the last six years, literature has covered implementing code clone detection tools using machine learning and deep learning. Lie et al. [12] reviewed this literature while surveying the application of deep learning on code clone detection between 2016 and 2020. They have found 21 papers that implemented deep learning in code clone detection software. While this is an exciting area, there are limitations concerning the scalability and performance of such techniques.

2.2 Tokenization

Grefenstette [4] defines tokenization as the first step in transforming text, and with each step, it abstracts away the surface differences. There is vast literature present on tokenization in the field. However, most of them are specific to the domain that tokenization is applied in. I will be covering the literature on text mining and information retrieval systems in this section. Vijayarani and Janani [28] define text mining as the extraction of interesting and non-trivial knowledge from unstructured text. Manning presented the book overviewing modern information retrieval. Ibrihicha [6] recently presented a review of research in Information Retrieval. They describe information retrieval as a fundamental idea that has found its place everywhere. Information retrieval systems are measured mainly using precision and recall.

2.3 Tokenization in Natural Language Processing

The intuition to have a literature survey for Natural Language Processing (NLP) is to understand how the tokenization process works for NLP models and take those insights for analyzing the tokenization parameters in the code clone detection tools. Webster and Kit [30] have examined the tokenization process in NLP by comparing the techniques used in Chinese and English. They conclude that it is essential to include more processes for tokenization than just using delimiters to find tokens. Mielke et al. [15] present a study of different ways of tokenization that include multi-word vocabulary building, byte-pair encoding, and subwords. They conclude that there is no singular silver bullet solution for tokenization; it will always be multiple approaches and dependent on the final model. Finally, Libovicky and Fraser [14] used the process of starting with subwords tokenization and then fine-tuning the character-wise process in NLP. While this process requires training a model different from SourcererCC, it presents an exciting approach of having the character-wise overlap similarity

in the clone detection algorithm.

Chapter 3

Methodology

My thesis focuses on the effects of parameters on the effectiveness of SourcererCC. This section is divided into the setup of tools along with the web interface, the recall and precision study required for measuring the effectiveness of SourcererCC, and additional concepts required to interpret experiments.

3.1 Setup

3.1.1 SourcererCC

SourcererCC presented in the original paper, can be run using the command line tool. In addition, source code can be cloned from the public GitHub repository. It uses multi-threading that is supported in Python 3.6. It consists of two steps, generating tokens and running the clone detector algorithm. Both these steps are executed using python scripts. First, tokens are generated that are stored with a partial index of the project files and the tokens in a file. This file is then used as input for the clone detector python script. The final

output is a CSV file with the clone pairs.

3.1.2 Web Interface of SourcererCC

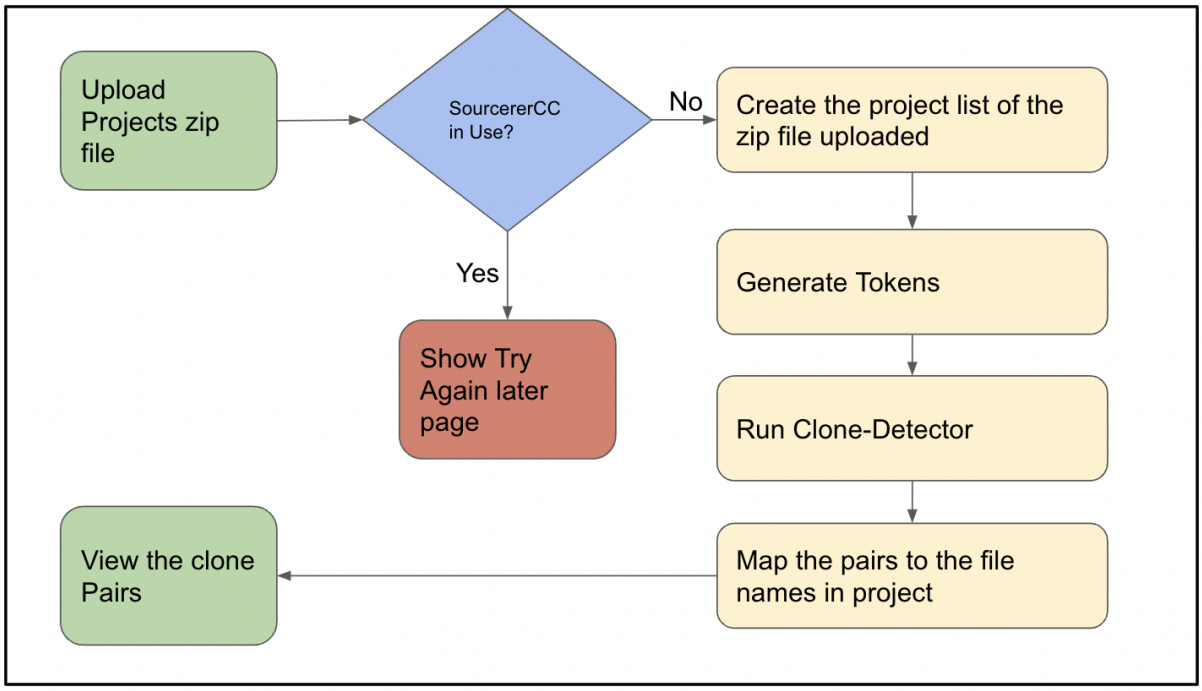

Figure 3.1: Website Workflow

I have created a web interface in python using flask [5] that only requires the user to upload the projects to be compared in a zip format. The first page (3.2) can be used to upload the project files. The website's backend implements both the steps the user requires to run in the command line. The website's workflow is shown in Figure 3.1. The final page (3.3) returns the results in the format of the result pairs with the actual project and file names for better understanding.

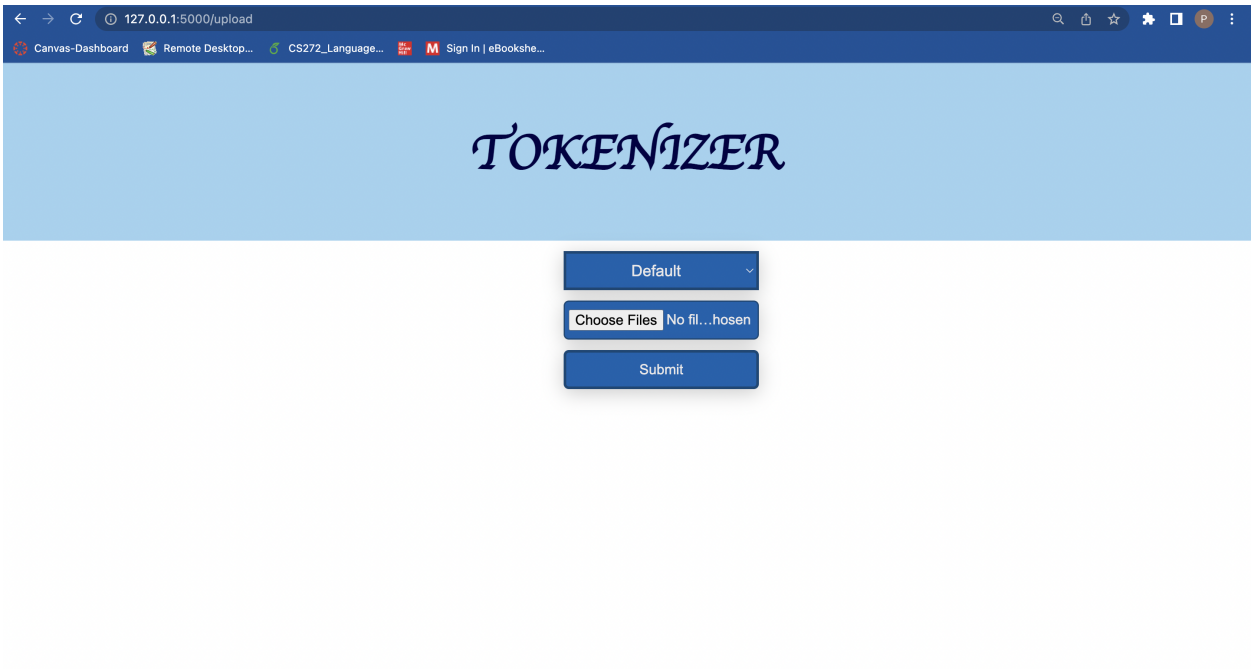

Figure 3.2: First page

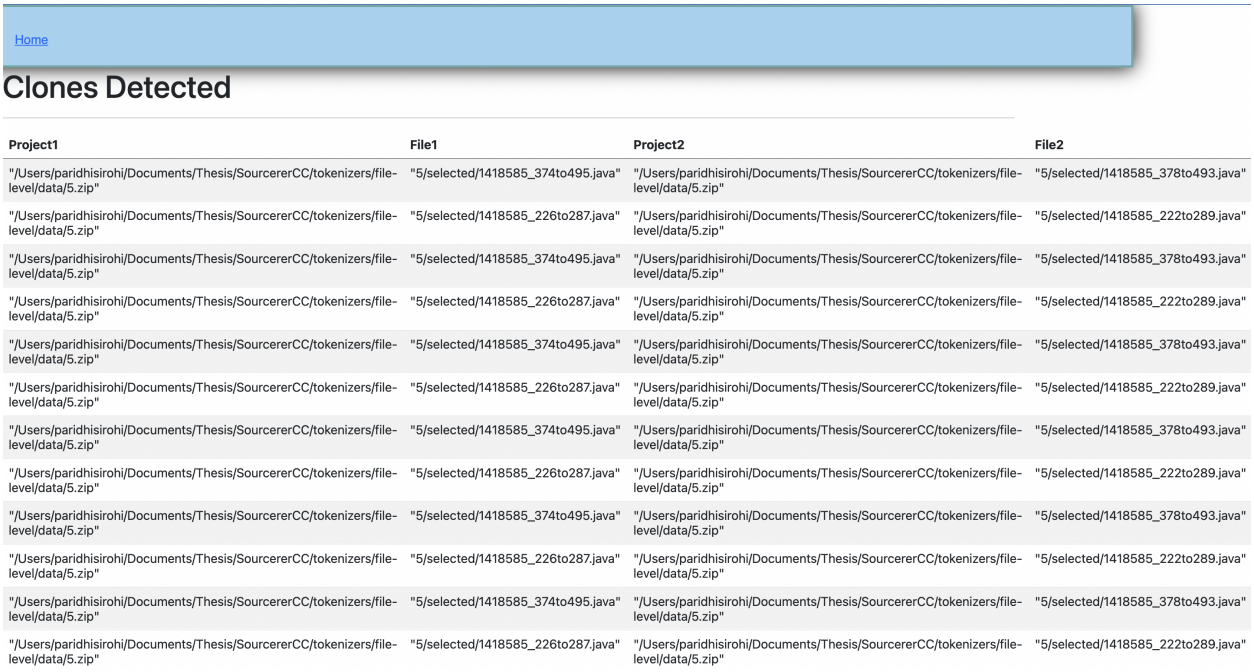

Figure 3.3: List of Clones

3.1.3 BigCloneBenchEval

The BigCloneEval [27] is a framework used for evaluating the recall of a code clone detection tool and uses the BigCloneBench database for evaluating the clones. The BigCloneBench database is a benchmark consisting of manually validated clone pairs from 25,000 java projects [26]. These java projects were taken from the inter-project Java repository IJa-Dataset, which consists of 8 million clone pairs covering 43 functionalities. This database has been used for evaluating the recall of various code clone detection tools. The BigCloneEval is used using the command line. The commands include initializing the framework, registering the tool, finding clones with the code clone detector tool, and evaluating the tool. For SourcererCC, clones are generated independently and imported into the BigCloneEval framework. The recommended configurations of the benchmark are to use minimum tokens of 50, minimum lines to be 6, and clone matcher of 70% threshold. I have used similar configurations. The distribution of the clone types is listed in Table 5.1.

Clone Type	1	2	Very Strongly 3	Strongly 3	Moderately 3
no of clone pairs	35787	4573	4156	14997	79756

Table 3.1: BigCloneEval Summary

3.1.4 Manual Precision study

There has been plenty of research and implementations of code clone detection tools. However, no tools are present for automating the precision evaluation of tools. In this thesis, I have used a manual study for the experiments. I have randomly selected 150 clones for each experiment from clone pairs generated from the reduced IJaDataset 2.0. The clones were distributed among four judges to reduce the bias in the final result. The four judges are clone experts and check the pairs for the same functionality.

3.2 Parameters

3.2.1 Threshold

The threshold for SourcererCC is set in the clone detector. The recall and precision are sensitive to the threshold of the tool. I have experimented with various thresholds with the original tokenizer to see the impact of the threshold on the clones detected.

3.2.2 Subtokens

The tokenization process for SourcererCC converts the files to tokens in a reduced form to increase the tool's scalability. I have used the naming conventions, the camel casing, and the underscore to generate subtokens.

3.2.3 Stop Words

During the tokenization of files for SourcererCC, there is a possibility to include the stop words in text format. All programming languages have a fixed set of keywords that will be commonly repeated. I have used this list as the stop words in the experiments.

3.3 Additional Context

3.3.1 Overlap Similarity

SourcererCC uses overlap similarity [11] for detecting clones. The intuition behind this thesis is to see how the selected experimental parameters can help increase the overlap similarity

boolean	byte	char	double	float
short	void	int	long	while
for	do	switch	break	continue
case	default	if	else	try
catch	finally	class	abstract	extends
final	import	new	instance of	private
interface	native	public	package	implements
protected	return	static	super	synchronized
this	throw	throws	transient	volatile

Figure 3.4: Java keywords [25]

for better clone detection. The overlap similarity can be measured with Jaccard coefficients [19] and cosine similarity.

Experiments conducted in this thesis can be done only on the file level. The BigCloneEval requires clones to be presented along with the pairs' start and end line numbers. To make the output of SourcererCC compatible with BigCloneEval, I have processed the dataset to convert it into functions with the start and end numbers in the file name for easy processing. I have used the python script presented in Figure 3.5.

```
def __get_start_end_for_node(node_to_find):
    start = None
    end = None
    for path, node in tree:
        if start is not None and node_to_find not in path:
            end = node.position
            return start, end
        if start is None and node == node_to_find:
            start = node.position
    return start, end


def __get_string(start, end):
    if start is None:
        return ""

    # positions are all offset by 1. e.g. first line -> lines[0], start.line = 1
    end_pos = None

    if end is not None:
        end_pos = end.line - 1

    lines = data.splitlines(True)
    global last_end
    last_end = len(lines)
    string = "".join(lines[start.line:end_pos])
    string = lines[start.line - 1] + string

    # When the method is the last one, it will contain a additional brace
    if end is None:
        left = string.count("{")
        right = string.count("}")
        if right - left == 1:
            p = string.rfind("}")
            string = string[:p]

    return string
```

Figure 3.5: Python Script to Split Files based on functions

Chapter 4

Evaluation

4.1 Experiment Procedure

For the results, I have conducted four experiments on SourcererCC. The first experiment is with the parameter threshold in the clone detector. The subsequent three experiments are conducted with a fixed threshold and done on the side of the tokenizer. The details of the experiment are covered in this section.

4.1.1 Threshold

For this experiment, I have varied the threshold during a preliminary study to see how it affects the clones detected. During this preliminary study, I hand-picked two code fragments given to SourcererCC with 70% and 80% thresholds. To further understand how the threshold affects the results, I evaluated the recall of Functionality 2 of BigCloneEval. The recall for both thresholds was 100% for Type 1 and 98% for Type 2. This shows that the use of a higher threshold did not have a significant effect on the recall. Also, a higher threshold

should help achieve better precision.

4.1.2 Fixed threshold with subtokens

I have chosen to conduct experiments on a fixed threshold for the following experiments. The original evaluation of SourcererCC was done with a 70% threshold. Intuitively, since the number of tokens will increase with subtokens. I have decided to experiment with an 80% threshold. The subtoken generation happens only on the file level.

4.1.3 Fixed threshold with stop words removal

Experiments are conducted over Java code fragments. The Java language uses 52 keywords. These keywords can be found listed in Figure 3.4. I have used the same 80% threshold for the stop words to maintain uniformity across the other experiments.

4.1.4 Fixed threshold with subtokens and stop words removal

The final experiment combines all three parameters identified in this thesis. A combination of stop words and subtokens is to understand how the two might affect each other in the clone detection process. The threshold is set to 80%.

4.2 Evaluation Metrics

I have used recall and precision to measure the quality of the clones generated by the experiments.

4.2.1 Recall

The ratio of the retrieved relevant records to the total number of relevant records is defined as recall. For code clone detection tools, these records are the clone pairs. I have used the BigCloneEval to evaluate the recall on the experiments covered in Section 4.1. The formula [20] is as follows:

$$\frac{NumberofClonesDetected}{NumberofClones}$$

Since the BigCloneEval consists only of valid clone pairs, the formula for the recall is just the number of clones the tool found vs. the total number of clones. The recall helps us understand the effectiveness of the parameters on SourcererCC.

4.2.2 Precision

The precision is the measure of how many results obtained are actually true. I have evaluated the precision by a manual study. I have taken 150 samples of each experiment and distributed them among clone experts to obtain the results. The formula [20] for the precision for clones is as follows:

$$\frac{NumberofTrueClones}{NumberofClones}$$

The effectiveness of SourcererCC is measured using two evaluation parameters, recall, and precision. Precision is used to understand how many clone pairs reported are valid clone

pairs. Recall checks how many clone pairs are correctly identified. Together, these two parameters help us measure the effectiveness of SourcererCC. The following section presents the results of the experiments discussed in terms of the evaluation metrics.

Chapter 5

Results and Findings

This section covers the results of the experiments conducted on SourcererCC. These experiments are conducted to evaluate the effectiveness of SourcererCC. Two chosen metrics, recall and precision, are discussed below:

5.1 Recall

The recall for the experiments was measured on the BigCloneEval. The results are presented in Table 5.1. The recall is reported for all clones, inter-project, and intra-project. I have set the minimum similarity to 50% while evaluating to target Types 1,2 and 3 mainly. The BigCloneEval evaluates the type 1, type 2, very strongly type 3 (VST3), strongly typed 3 (ST3), and moderately typed 3 clones. The recall for the Type 1,2 and VST3 are between 97-100 % leaving less scope for analysis.

To understand how it affects Type 1 and Type 2 for comparison, I have removed the datasets for functionalities 31, 34, 41, and 44. These results are only for comparison purposes. The

Experiment	All Clones					Intra-Project					Inter-Project				
	T1	T2	VST3	ST3	MT3	T1	T2	VST3	ST3	MT3	T1	T2	VST3	ST3	MT3
Section 4.1.1 70%	100	98	93	61	5	100	99	99	86	14	100	97	86	48	5
Section 4.1.1 80 %	100	97	91	57	2	100	98	98	83	10	100	96	83	42	3
Section 4.1.2	100	99	94	59	5	100	99	99	85	13	100	98	85	46	5
Section 4.1.3	100	96	90	55	1	100	98	97	82	9	100	96	82	41	3
Section4.1.4	100	99	95	56	5	100	99	98	84	11	100	97	84	42	5

Table 5.1: BigCloneEval Recall

results are captured in Table 5.2. In the first experiment (Section 4.1.1), 70% and 80% resulted in Type 1 and Type 2 clones. For type 3 clones, the recall values reduced for threshold 80%. Since SourcererCC uses overlap similarity, this effect is expected to increase the threshold.

For experiment 2 (Section 4.1.2), I received the best results for recall. Type 1 and Type 2 perform 9% and 5% better. However, since the code clone detection tools are very susceptible to configurations, it is possible to achieve the maximum result for all the experiments in the case of Type 1 and Type 2. For the Type 3 clones, it increases the recall for VST3, but similar trends are not observed with ST3 and MT3. This trend is justifiable as only the overlap similarity increases, but that does not affect these types of clones.

For experiment 3 (Section 4.1.3), the recall metric reflects our hypothesis in the preliminary results. As the standard tokens reduce but the threshold is high, the experiment performs worse than the original SourcererCC. This issue is combated in the fourth experiment. The experiment 4 (Section 4.1.4) configuration affects the VST3 clones the most by showing an increase of 10% from the original results.

Experiment	All Clones					Intra-Project					Inter-Project				
	T1	T2	VST3	ST3	MT3	T1	T2	VST3	ST3	MT3	T1	T2	VST3	ST3	MT3
Section 4.1.1 70%	50	72	42	30	1	74	78	42	16	1	45	39	42	57	6
Section 4.1.1 80%	50	72	41	24	0	74	78	42	13	0	45	38	39	46	2
Section 4.1.2	59	77	54	25	1	87	81	66	22	1	54	54	43	33	35
Section 4.1.3	48	71	36	11	0	72	78	41	8	0	44	34	32	17	1
Section 4.1.4	58	74	51	21	1	85	79	63	17	1	53	43	42	28	2

Table 5.2: BigCloneEval Recall without Functionalities 31,34,41,44

5.2 Precision

The results are reported in Table 5.3. The precision for threshold 70 (Section 4.1.1) was reported in the original paper as 91%. I have achieved a precision of 91 % with a threshold of 80 (Section 4.1.1). The precision for the subtokens (Section 4.1.2) experiment matches the expectations of it being lower than the original experiment. However, the precision for the stop words experiment (Section 4.1.3) was also lower. One possible reason can be that the study was done on Java files, and Java is a syntactically typed language. The removal of keywords has resulted in reducing the overlap of the tokens. This reduction shows that stop words alone do not help with precision. I have achieved 91% precision in the stop words and subtokens experiment (Section 4.1.4).

Experiment	Precision
Section 4.1.1	91
Section 4.1.2	87
Section 4.1.3	84
Section 4.1.4	91

Table 5.3: Precision

During the study, all the judges noticed that the false positives in the case of experiment 4 (Section 4.1.4) were the hardest to find as the clones were syntactically similar. However, the difference was in the actual functionality of code fragments. This issue falls outside the domain of SourcererCC as it is a syntactically dependent clone detection tool.

Chapter 6

Conclusion and Future Work

In this thesis, I have looked at the interactions between the various parameters that affect the metrics for code clone detection tools.

Using a web interface for code clone detection tools will benefit everyone. It can also help by providing uniformity across different studies. I conclude that subtokens and stop words help achieve better precision and recall. However, the use of subtokens also increases false positive results. The use of stop words helps combat the precision loss caused by the subtokens. Also, the two configurations together help achieve better results.

6.1 Threats to Validity

I have identified the following threats to our approach:

- Recall Study: The difference in configurations helps attain close to 100% recall for Type 1 and Type 2. While this is a comparative study, the experiment is conducted to show how it differentiates; using parameters might not be helpful for a system already

achieving 100

- Manual sampling of Precision Study: The samples are randomly selected but can only reflect one section of the dataset. Since the datasets are vast, the actual precision might be different from the one observed in this study.

6.2 Future Work

The future work for this thesis can include experimentation in the direction of NLP models. A few concepts, such as building vocabulary and word embeddings, can be used to understand how it affects the code clone detection tools. In addition, this direction can help identify Type 4 pairs that work on the same functionality.

I have experimented with the SourcererCC. Other clone detection tools available in the literature may behave differently. Experimentation with other code clone detection tools is one direction in which affect of configurations on code clone detection tools can be evaluated. These studies will also help build an understanding of how the evaluation of tokenization can be done.

Bibliography

[1] B. Baker. On finding duplication and near-duplication in large software systems. In *Proceedings of 2nd Working Conference on Reverse Engineering*, pages 86–95, 1995.

[2] S. Ducasse, M. Rieger, and S. Demeyer. A language independent approach for detecting duplicated code. In *Proceedings IEEE International Conference on Software Maintenance - 1999 (ICSM'99). 'Software Maintenance for Business Change' (Cat. No.99CB36360)*, pages 109–118, 1999.

[3] S. Gangappa, A. Rana, V. Kansal, and S. Tanwar. *Code Clone Detection—A Systematic Review*, pages 645–655. 05 2021.

[4] G. Grefenstette. *Tokenization*, pages 117–133. Springer Netherlands, Dordrecht, 1999.

[5] M. Grinberg. *Flask web development: developing web applications with python*. " O'Reilly Media, Inc.", 2018.

[6] S. Ibrihich, A. Oussous, O. Ibrihich, and M. Esghir. A review on recent research in information retrieval. *Procedia Computer Science*, 201:777–782, 2022.

[7] E. Juergens, F. Deissenboeck, and B. Hummel. Clonedetective - a workbench for clone detection research. In *2009 IEEE 31st International Conference on Software Engineering*, pages 603–606, 2009.

[8] T. Kamiya, S. Kusumoto, and K. Inoue. Ccfinder: a multilinguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering*, 28(7):654–670, 2002.

[9] S. Kawaguchi, T. Yamashina, H. Uwano, K. Fushida, Y. Kamei, M. Nagura, and H. Iida. Shinobi: A tool for automatic code clone detection in the ide. In *2009 16th Working Conference on Reverse Engineering*, pages 313–314. IEEE, 2009.

[10] R. Komondoor and S. Horwitz. Tool Demonstration: Finding Duplicated Code Using Program Dependences. In D. Sands, editor, *Proceedings of the 10th European Symposium on Programming*, volume 2028 of *Lecture Notes in Computer Science*, pages 383–386. Springer-Verlag, 2001.

[11] L. R. Lawlor. Overlap, similarity, and competition coefficients. *Ecology*, 61(2):245–251, 1980.

[12] M. Lei, H. Li, J. Li, N. Aundhkar, and D.-K. Kim. Deep learning application on code clone detection: A review of current knowledge. *Journal of Systems and Software*, 184:111141, 2022.

[13] Z. Li, S. Lu, S. Myagmar, and Y. Zhou. Cp-miner: finding copy-paste and related bugs in large-scale software code. *IEEE Transactions on Software Engineering*, 32(3):176–192, 2006.

[14] J. Libovickỳ and A. Fraser. Towards reasonably-sized character-level transformer nmt by finetuning subword systems. *arXiv preprint arXiv:2004.14280*, 2020.

[15] S. J. Mielke, Z. Alyafeai, E. Salesky, C. Raffel, M. Dey, M. Gallé, A. Raja, C. Si, W. Y. Lee, B. Sagot, et al. Between words and characters: A brief history of open-vocabulary modeling and tokenization in nlp. *arXiv preprint arXiv:2112.10508*, 2021.

[16] F. P. Miller, A. F. Vandome, and J. McBrewster. Camelcase: Compound (linguistics), whitespace (computer science), capitalization, patti labelle, visual basic, macgyver, ipod, chemical formula, naming... programming language, marketing, 2009.

[17] M. Mondal, C. K. Roy, M. S. Rahman, R. K. Saha, J. Krinke, and K. A. Schneider. Comparative stability of cloned and non-cloned code: An empirical study. In *Proceedings of the 27th Annual ACM Symposium on Applied Computing*, SAC '12, page 1227–1234, New York, NY, USA, 2012. Association for Computing Machinery.

[18] H. Murakami, K. Hotta, Y. Higo, H. Igaki, and S. Kusumoto. Folding repeated instructions for improving token-based code clone detection. In *2012 IEEE 12th International Working Conference on Source Code Analysis and Manipulation*, pages 64–73. IEEE, 2012.

[19] S. Niwattanakul, J. Singthongchai, E. Naenudorn, and S. Wanapu. Using of jaccard coefficient for keywords similarity. In *Proceedings of the international multiconference of engineers and computer scientists*, volume 1, pages 380–384, 2013.

[20] D. L. Olson and D. Delen. *Advanced data mining techniques*. Springer Science & Business Media, 2008.

[21] D. Rattan, R. Bhatia, and M. Singh. Software clone detection: A systematic review. *Information and Software Technology*, 55(7):1165–1199, 2013.

[22] C. Roy and J. Cordy. A survey on software clone detection research. *School of Computing TR 2007-541*, 01 2007.

[23] C. K. Roy, J. R. Cordy, and R. Koschke. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Science of Computer Programming*, 74(7):470–495, 2009.

[24] H. Sajnani, V. Saini, J. Svajlenko, C. K. Roy, and C. V. Lopes. Sourcerercc: Scaling code clone detection to big-code. In *Proceedings of the 38th International Conference on Software Engineering*, pages 1157–1168, 2016.

[25] H. Schildt and D. Coward. *Java: the complete reference*. McGraw-Hill Education New York, 2014.

[26] J. Svajlenko and C. K. Roy. Evaluating clone detection tools with bigclonebench. In *2015 IEEE international conference on software maintenance and evolution (ICSME)*, pages 131–140. IEEE, 2015.

[27] J. Svajlenko and C. K. Roy. Bigcloneeval: A clone detection tool evaluation framework with bigclonebench. In *2016 IEEE international conference on software maintenance and evolution (ICSME)*, pages 596–600. IEEE, 2016.

[28] S. Vijayarani, R. Janani, et al. Text mining: open source tokenization tools-an analysis. *Advanced Computational Intelligence: An International Journal (ACII)*, 3(1):37–47, 2016.

[29] T. Wang, M. Harman, Y. Jia, and J. Krinke. Searching for better configurations: A rigorous approach to clone evaluation. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, page 455–465, New York, NY, USA, 2013. Association for Computing Machinery.

[30] J. J. Webster and C. Kit. Tokenization as the initial phase in nlp. In *COLING 1992 volume 4: The 14th international conference on computational linguistics*, 1992.

[31] M. Ďuračík, E. Kršák, and P. Hrkút. Current trends in source code analysis, plagiarism detection and issues of analysis big datasets. *Procedia Engineering*, 192:136–141, 2017. 12th international scientific conference of young scientists on sustainable, modern and safe transport.