DETECTION AND ANALYSIS OF SOFTWARE CLONES

By

ABDULLAH MOHAMMAD SHENEAMER

M.S., University of Colorado Colorado Springs, Computer Science, USA, 2012
B.S., University of King Abdulaziz, Computer Science, KSA, 2008

A dissertation submitted to the Graduate Faculty of the

University of Colorado Colorado Springs

in partial fulfillment of the

requirement for the degree of

Doctor of Philosophy

Department of Computer Science

2017
This dissertation for Doctor of Philosophy degree by
Abdullah Mohammad Sheneamer
has been approved for the
Department of Computer Science
by

Jugal Kalita, Chair

Tim Chamillard

Albert Glock

Jonathan Ventura

Charlie Wang

07/21/2017

Date
ABSTRACT

Effective detection of code clones is important for software maintenance. Code clones introduce difficulty in software maintenance and lead to bug propagation. Detection of duplicated bugs within a piece of software is challenging, especially when duplications are semantic in nature, where textually two pieces of code are different although they perform the same task. Similar issues can also be observed in malware detection or more precisely, obfuscated code detection.

In this dissertation, we first conduct a comprehensive study on state-of-the-art clone detection tools and report an empirical comparative analysis of different methods.

Next, we propose a new hybrid clone detection technique. It is a two-step process. First, it uses a coarse grained technique to analyze clones effectively to improve precision. Subsequently, it uses a fine-grained detector to obtain additional information about the clones and to improve detection accuracy of Type-I, Type-II and Type-III clones.

The task of clone detection is more challenging when clones are semantically similar in nature, but have no textual resemblance to each other. We present a novel machine learning framework for automated detection of all four types of clones using features extracted
from Abstract Syntax Trees (ASTs) and Program Dependency Graphs (PDGs), from pairs of code blocks.

Majority of publicly available clone data sets are incomplete in nature and lack labeled samples of Type-IV. It makes difficult for any machine learning framework using such datasets to be useful. In our third contribution, we propose a new scheme for labeling semantic code clones or Type-IV clones. We introduce a new dataset of clone references, which is a set of correct Type-IV clones. This contribution can help researchers evaluate techniques that detect cloned code of Type-IV.

Code obfuscation is a technique to alter the original content of the code to confound reverse engineering. Obfuscated code detection is challenging due to the availability of code obfuscation tools. We observe a resemblance between semantic clones and obfuscated code. We apply our clone detection scheme to detect obfuscated code. We propose a framework that can detect both code clones and obfuscated code as our final contribution. Our results are far superior in comparison to state-of-the-art obfuscated code detection methods.
To my parents, beloved wife, my brothers, my sisters,

and my lovely child.
ACKNOWLEDGMENTS

I would like to express my deepest appreciation to my wife and the mother of my child Hanan for the unlimited support she gave to me during my journey toward my Ph.D. I am also very grateful to the support and guidance provided by my advisor Professor Jugal Kalita. He gave me the freedom to explore new research ideas. In addition, I would like to thank my dissertation committee members: Dr. Tim Chamillard, Dr. Albert Glock, Dr. Jonathan Ventura and Dr. Charlie Wang for their guidance and consultation. I would like also to express my sincere gratitude to Dr. Swarup Roy for being willing to join my research and give beneficial feedback. During the course of my Ph.D., many close friends and family members stood by me through tough times. I am grateful to all my friends for their support. Finally, I would like to thank my wonderful parents, who have made countless sacrifices for my well-being and education. They have sacrificed their life to see me at where I am today. I wish I could learn to love the way they love me. I dedicate this dissertation to them.

Funding Acknowledgment. I was supported by Jazan University, place, country for the most of time I was working on this dissertation.
# TABLE OF CONTENTS

## CHAPTER

### I INTRODUCTION 1

- Overview ......................................................... 1
- Basic Definitions ............................................ 2
- Types of Clones .................................................. 3
- Clone Detection Phases ......................................... 4
- Code Preprocessing or Lexical Analysis ...................... 4
- Transformation .................................................... 5
- Match Detection ................................................... 7
- Formatting ........................................................... 7
- Filtering ............................................................. 7
- Aggregation .......................................................... 7
- Assessment Metrics ............................................... 8
- Precision and Recall ............................................. 8
- Portability .......................................................... 10
- Scalability .......................................................... 10
- Comparison Units .................................................. 10
- Robustness .......................................................... 11
- Language Independence .......................................... 11
- Types of Clones ................................................... 11
- Application of Clone Detection ................................. 11
Clone Avoidance ................................................. 12
Plagiarism Detection ........................................... 12
Code Obfuscation Detection ................................. 12
Bug Detection .................................................. 13
Code Compacting .............................................. 13
Copyright Infringement ....................................... 13
Clone Detection in Models ................................. 14
Motivation and Main Contributions ..................... 14
Thesis Outline .................................................. 16
Publication List ............................................... 18

II LITERATURE REVIEW 19
Introduction .................................................... 19
Categories of Detection Techniques .................... 20
Textual Approaches .......................................... 20
Lexical Approaches .......................................... 29
Syntactical Approaches .................................... 38
Semantic Approaches ...................................... 48
Chapter Summary ........................................... 55

III CODE CLONE DETECTION USING A COARSE AND FINE-GRAINED HY-
BRID APPROACH 59
Introduction .................................................... 59
The Proposed Method ....................................... 60
Lexical Analysis and Normalization .......................... 63
Detecting Blocks and Extracting Sub-Blocks .............. 63
Grouping and Hashing Normalized Blocks .................. 63
Detecting Similar Blocks Using Levenshtein Distance/Cosine Similarity ... 64
Identifying Gapped Lines .................................... 64
Mapping Similar Blocks to the Original Source Code ........ 65
Similarity Measures .......................................... 65
Levenshtein Similarity ....................................... 66
Cosine Similarity .............................................. 66
Experiment Design .............................................. 67
Experiments ....................................................... 69
Experiment A ..................................................... 73
Experiment B ..................................................... 74
Discussions ....................................................... 77
Clone References ............................................... 77
Hashing Collision .............................................. 77
Different Programming Languages in Target Software Systems ............ 77
Thresholds for Leveshtien Distance and Cosine Similarity ............... 78
Chapter Summary .............................................. 79

IV MACHINE LEARNING FRAMEWORK FOR DETECTING SEMANTIC CODE

CLONES .......................................................... 82
Introduction ...................................................... 82
An Integrated Detection Framework ................................................. 124
Java ByteCode : Low Level Features .............................................. 126
Source Code Features .................................................................. 129
Fusion of Code Features ................................................................ 131
Code Similarity as a Feature .......................................................... 132
A New Code Obfuscation and Clone Detection Scheme ...................... 133
Experimental Evaluation ................................................................. 134
Datasets ......................................................................................... 136
Ensemble Classification Model ........................................................ 138
Experimental Results .................................................................... 139
Performance comparison ................................................................. 143
Chapter Summary ......................................................................... 147

VII CONCLUSION AND FUTURE WORK .......................................... 149
Conclusion ...................................................................................... 149
Future Work ................................................................................. 150

REFERENCES ................................................................................. 153

APPENDICES

Appendix A FEATURES AND RESULTS OF CHAPTER IV .................. 168
   Traditional Features ................................................................. 168
   AST Features ............................................................................. 169
   PDG Features ........................................................................... 170
Results of Combinations of Pair Instances Vectors . . . . . . . . . . . . . . . 171

Results of BigCloneBench Recall, Precision and F-Measure measurements . 172

Appendix B  FEATURES USED IN CHAPTER VI  173

More AST Features . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 173

BDG Features . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 174
LIST OF TABLES

TABLE

2.1 Techniques for clone detection. ........................................... 21
2.2 Summary of textual approaches ......................................... 22
2.3 Summary of lexical approaches ......................................... 30
2.4 Summary of syntactical approaches ................................. 39
2.5 Summary of semantic approaches ................................. 50
2.6 Evaluations of clone detection tools through the dissertation ............... 56
3.1 Target software systems ............................................ 67
3.2 Comparison the medians and the averages of our approaches for Type-III clones results with other existing approaches. ......................... 76
3.3 Comparison the medians and the average of our approaches for all Type-I, II and III clones results with other existing approaches. ......................... 76
4.1 Classification techniques used ........................................... 101
5.1 Brief description of our Java code clone corpus ....................... 116
6.1 ByteCode conditional statements ....................................... 127
6.2 Categorization of ByteCode instructions ....................................... 127
6.3 Brief description of Java obfuscated datasets ....................... 138
LIST OF FIGURES

FIGURE

1.1 Simple examples of types of clones. ........................................... 3
1.2 Four phases in CCFinder clone detection tool. .......................... 5

2.1 CP-Miner process steps. ..................................................... 33
2.2 Example of coarse-grained clone detection. ............................. 43

3.1 The proposed code clone detection method. Each step is illustrated as we analyze the code in two files for the existence of clones. .............. 61
3.2 The results for Type-I and Type-II. ....................................... 70
3.3 The results for Type-III. .................................................... 71
3.4 The results for Type-I, II and III. ........................................ 72
3.5 Cosine similarity speed. .................................................... 75
3.6 Levenshtein distance speed. ............................................... 76
3.7 Levenshtein distance threshold. ......................................... 78
3.8 Cosine threshold. .......................................................... 79

4.2 Program dependency graph showing control and data dependency among the statements. ......................................................... 94
4.3 Share of different categories of features used. ......................... 96
4.4 Workflow of the proposed clone detection framework. ............... 99
4.5 Performance all the candidate classifiers with different feature fusions. . 103
4.6 Performance of three best classifiers with syntactic and semantic features. . 104
4.7 Learning curve: Performance of Random Forest and Xgboost with varying features. ................................................................. 105
4.8 Performance of Random Forest and Rotation Forest with varying number of features using Gain Ratio and InfoGain feature selection algorithms. . . . . . . . . . . 106
4.9 Performance comparison of different detection methods with respect to different assessment metrics. ................................................................. 107

5.1 A new framework for labeling semantic code clones. ................................. 114

6.1 A BDG showing control and data dependency among instructions. ........ 127
6.2 Share of categories of features used. ................................................................. 133
6.3 Workflow of the proposed dual detection framework. ................................. 135
6.4 Effectiveness of the framework for detecting code clones when the training and test datasets are from different corpora. ................................. 136
6.5 Java obfuscated dataset generation steps. ................................................................. 138
6.6 Effectiveness of the framework for detecting obfuscated code using a model that was built by training on one dataset and then tested on another. ........ 139
6.7 Performance of detection framework on clone datasets with different feature combinations. ................................................................. 140
6.8 Effectiveness of the framework on detecting obfuscated code using feature fusions. ................................................................. 140
6.9 Learning curve: Performance of ensemble classifier on clone dataset with varying number of features. ................................................................. 141
6.10 Performance of ensemble approach on clone dataset after feature selection using MDI. ................................................................. 142
6.11 Effectiveness of ensemble classifier for detecting obfuscated code after feature selection using MDI. ................................................................. 142
6.12 Prediction effectiveness of proposed framework in comparison to state-of-the-art clone detectors in terms of F-Score. .......................... 144

6.13 Effectiveness of various obfuscated code detection tools on ObsCode dataset. 146

6.14 Tool performance comparison on the PacMan data in terms of accuracy, in which the original program is compared to obfuscated programs for each clone detection technique. ........................................ 147
CHAPTER I

INTRODUCTION

Overview

If two fragments of source code are identical or similar to each other, they are called code clones. Software clones occur due to reasons such as code reuse by copying pre-existing fragments, coding style, and repeated computation using duplicated functions with slight changes in variables or data structures used. If we edit a code fragment, it will have to be checked against all related code clones to see if they need to be modified as well. Removal, avoidance or refactoring of cloned code are other important issues in software maintenance. However, several research studies have demonstrated that removal or refactoring of cloned code is sometimes harmful [94].

Code clones are used frequently because they can be created fast, and easily inserted with little expense [112]. However, code clones affect software maintenance, may introduce poor design, lead to wasted time in repeatedly understanding a fragment of poorly written code, increase system size, and reincarnate bugs that are present in the original code segment. All these make it a difficult job to maintain a large system [74, 86]. Additional
reasons that make code clone detection essential are the following. 1) Detecting cloned code may help detect malicious software [104]. 2) Code clone detection may find similar code and help detect plagiarism and copyright infringement [49, 74, 104]. 3) Code clone detection helps reduce the source code size by performing code compaction [17]. 4) Code clone detection also helps detect crosscutting concerns, which are aspects of a program that impact other issues that arise when code is duplicated all over the system [14].

**Basic Definitions**

Each paper in the literature defines clones in its own way [86]. Here, we provide common definitions which we use throughout our thesis.

**Definition 1: Code Fragment.** A code fragment (CF) is a part of the source code needed to run a program. It usually contains more than five lines that are considered interesting, but it may contain fewer than five lines. It can contain a function or a method, begin-end blocks or a sequence of statements [11, 94].

**Definition 2: Software Clone/Code Clone/Clone Pair.** If a code fragment $CF_1$ is similar to another code fragment $CF_2$ syntactically or semantically, one is called a clone of the other. If there is a relation between two code fragments such that they are analogous or similar to each other, the two are called a clone pair ($CF_1, CF_2$).

**Definition 3: Clone Class.** A clone class is a set of clone pairs where each pair is related by the same relation between the two code fragments. A relation between two code fragments is an equivalence relation which is reflexive, symmetric and transitive, and holds between two code fragment if and only if they are the same sequence.
Figure 1.1: Simple examples of types of clones.

Types of Clones

There are two groups of clones. The first group refers to two code fragments which are similar based on their text [10,53]. There are three types within the first group as shown in Figure 1.1.

Type-I (Exact clones): Two code fragments are the exact copies of each other except whitespaces, blanks and comments.

Type-II (Renamed/Parameterized): Two code fragments are similar except for names of
variables, types, literals and functions.

**Type-III (Near miss clones/Gapped clones):** Two copied code fragments are similar, but with modifications such as added or removed statements, and the use of different identifiers, literals, types, whitespaces, layouts and comments.

The second group refers to two code fragments which are similar based on their functions [35]. Such clones are also called Type-IV clones as shown in Figure 1.1.

**Type-IV (Semantic clones):** Two code fragments are semantically similar, without being syntactically similar.

### Clone Detection Phases

A clone detector is a tool that reads one or more source files and finds similarities among fragments of code or text in the files. Since a clone detector does not know where the repeated code fragments occur in advance, it must compare all fragments to find them. There are many previous proposed techniques that perform the necessary computation and attempt to reduce the number of comparisons.

We first discuss the phases of clone detection in general. A clone detection technique may focus on one or more of the phases. The first four of phases are shown in Figure 1.2

**Code Preprocessing or Lexical Analysis**

This process removes uninteresting pieces of code, converts source code into units, and determines comparison units. The three major purposes of this phase are given below.
Figure 1.2: Four phases in *CCFinder* clone detection tool [49].

1. **Remove uninteresting pieces of code.** All elements in the source code that have no bearing on the comparison process are removed or filtered out in this phase.

2. **Identify units of source code.** The rest of the source code is divided into separate fragments, which are used to check for the existence of direct clone relations to each other. Fragments may be files, classes, functions, *begin-end* blocks or statements.

3. **Identify comparison units.** Source units can be divided into smaller units depending upon the comparison algorithm. For example, source units can be divided into tokens.

**Transformation**

This phase is used by all approaches except text-based techniques for clone detection. This phase transforms the source code into a corresponding intermediate representation for comparison.
There are various types of representations depending on the technique. The usual steps in transformation are given below.

1. **Extract Tokens.** Tokenization is performed during lexical analysis by compiler front ends in programming languages [6, 45, 49, 66]. Each line of source code is converted into a sequence of tokens. In this step, all whitespaces, blanks and comments are removed. There is no comparison between the tokens in this step.

2. **Extract Abstract Syntax Tree.** All of the source code is parsed to convert into an abstract syntax tree or parse tree for subtree comparisons [10, 103].

3. **Extract PDG.** A Program Dependency Graph (PDG) represents control and data dependencies. The nodes of a PDG represent the statements and conditions in a program. Control dependencies represent flow of control information within the program. Data dependencies represent data flow information in a program. A PDG is generated by semantics-aware techniques from the source code for sub-graph comparison [57].

4. **Other Transformations.** Some techniques apply transformation rules to the source code elements before proceeding with clone detection. These include the CCFinder tool by Kamiya et al. [49], which has transformation rules for C++ templates for removing parameters. An example rule is $Name \langle ParameterList \rangle \rightarrow Name$. For example, $foo<int>$ is transformed into $foo$ [49].

5. **Normalization.** This step for removing differences is optional. Some tools perform normalization during transformation. This involves removing comments, whitespaces and differences in spacing as well as normalizing identifiers.
Match Detection

The result of transformation or normalization is the input to this phase. Every transformed fragment of code is compared to all other fragments using a comparison algorithm to find similar source code fragments. The output is a set of similar code fragments either in a clone pair list or a set of combined clone pairs in one class or one group as shown in Figure 1.2. For example, each clone pair may be represented as a quadruplet \((L\text{Begin}, L\text{End}, R\text{Begin}, R\text{End})\), where \(L\text{Begin}\) and \(L\text{End}\) are the left beginning and ending positions of a clone, and \(R\text{Begin}\) and \(R\text{End}\) are the right beginning and ending positions of another clone that following a clone pair [49].

Formatting

This step converts a clone pair list obtained by the comparison algorithm in the previous step into a new clone pair list related to the original source code.

Filtering

Not all clone detectors perform this step. In this phase, code clones are extracted and a human expert filters out the false-positive clones. This step is called Manual Analysis [93]. The false positives can also be filtered out by automated heuristics based on length, diversity or frequency.

Aggregation

This phase is optional. It can be done in the Match Detection phase. To reduce the amount of data, clone pairs can be aggregated into clusters, groups, sets or classes. For
example, clone pairs \((C_1, C_2), (C_1, C_3), \) and \((C_2, C_3)\) can be combined into the clone group \((C_1, C_2, C_3)\).

**Assessment Metrics**

In order to choose the right technique for a specific task, several evaluation metrics can be used. A good technique should show both high recall and precision. In Table 2.6, we provide a comparison of tools and evaluation approaches. Some evaluation metrics are discussed below.

**Precision and Recall**

Precision and recall are the two most common metrics used to measure the quality of a clone finding program. Precision refers to the fraction of candidate clones returned by the detection algorithm that are actual clones, whereas recall refers to the fraction of relevant candidate clones returned by the detection algorithm. High precision means that the candidate clones are mostly actual code clones. Low precision means that many candidate clones are not real code clones. High recall means most clones in the software have been found. Low recall means most clones in the software have not been found. Precision and recall are calculated as shown in Figure 7 and Equations (8) and (9) respectively:

\[
Precision = \frac{CC}{AC} \times 100
\]  

\[\text{(1.1)}\]
Recall \( = \frac{CC}{PC} \times 100 \) \hspace{1cm} (1.2)

where \( CC \) is the number of all correct clones, \( AC \) is the number of all found clones, and \( PC \) is the number of clones that exist in the code. A perfect clone detection algorithm has recall and precision values that are both 100%.

**Precision.**

A good tool detects fewer false positives, which means high precision. Line-based techniques detect clones of Type-I with high precision. There are no returned false positives and the precision is 100%. In contrast, token-based approaches return many false positives because of transformation and/or normalization. Tree-based techniques detect code clones with high precision because of structural information. Metric-based techniques find duplicated code with medium precision due to the fact that two code fragments may not be the same but have similar metric values. Finally, PDG-based techniques detect duplicated code with high precision because of both structural and semantic information.

**Recall.**

A good technique should detect most or all of the duplicated code in a system. Line-based techniques find only exact copy or Type-I clones. Therefore, they have low recall. Token-based techniques can find most clones of Type-I, Type-II and Type-III. So, they have high recall. A tree-based technique does not detect any type of clones, but with the help of other techniques clones can be detected. Metric-based techniques have low recall whereas PDG-based techniques cannot detect all of clones.
**Portability**

A portable tool is good for multiple languages and dialects. Line-based techniques have high portability but need a lexical analyzer. Token-based techniques need lexical transformation rules. Therefore, they have medium portability. Metric-based techniques need a parser or a $PDG$ generator to generate metric values. They have low portability. Finally, $PDG$-based techniques have low portability because they need a $PDG$-generator.

**Scalability**

A technique should be able to detect clones in large software systems in a reasonable time using a reasonable amount of memory. Scalability of text-based and tree-based techniques depends on the comparison algorithms. Token-based techniques are highly scalable when they use a suffix-tree algorithm. Metric-based techniques are also highly scalable because only metric values of $begin-end$ blocks are compared. $PDG$-based techniques have low scalability because subgraphs matches are expensive.

**Comparison Units**

There are various levels of comparison units such as source lines, tokens, subtrees and subgraphs. Text-based techniques compare the source code line-by-line, but their results may not be meaningful syntactically. Token-based techniques use tokens of the source code. However, token-based techniques can be less efficient in time and space than text-based techniques because a source line may contain several tokens. Tree-based techniques use tree nodes for comparison units and search for similar trees with expensive comparison,
resulting in low recall. Metric-based techniques use metric values for each code fragment but it could be that the metric values for cloned code are not the same. PDG-based techniques use PDG nodes and search for isomorphic subgraphs but graph matching is costly.

**Robustness**

When a tool can detect different clone types with higher precision and recall, is called a robust code clone detector. The right tool or approach should provide high precision and recall with all clone types.

**Language Independence**

A language independent tool is able to be work on any system without any issues. Thus, we should be aware of any language-dependent issues for our chosen method.

**Types of Clones**

In general, there are only four types of clones. Some approaches detect Type-I clones while others find Type-I, Type-II or Type-III clones or may even detect all types of clones.

**Application of Clone Detection**

Code clone detection techniques can help in areas such as clone refactoring or removal, clone avoidance, plagiarism detection, bug detection, code compacting, copyright infringement detection, in addition to clone detection.
Clone Avoidance

Two approaches are usually discussed in the context of cloning, how to detect clones and how to remove clones. The third approach is avoidance, which tries to disallow the creation of code clones in the software right from the beginning. Legue et al. [61] use code clone detection tools in two ways in software development. The first way uses code clone detection as preventive control where any added code fragment is checked to see if it is a duplicated version of any existing code fragment before adding it to the system. The second way, problem mining, searches for the modified code fragment in the system for all similar code fragments.

Plagiarism Detection

Code clone detection approaches can be used for plagiarism detection of software code. Dup [6] is a technique that is used for finding near matches of long sections of software code. JPlag [81] is another tool that finds similarities among programs written in C, C++, Java and Scheme. JPlag compares bytes of text and program structure. Yuan et al. [114] propose a count-based clone detection technique called CMCD. CMCD has been used to detect plagiarism in homeworks of students.

Code Obfuscation Detection

Code obfuscation is a technique to alter original content of the code in order to confuse the outside world. Malware creators use obfuscation for camouflaging existing ma-
licious code and make the task of signature based malware detection tool more challenging [19].

The semantics of a piece of obfuscated program is difficult to characterize. A number of methods and software tools are available for detecting obfuscated code. In Chapter VI, we propose a novel integrated framework for detecting both Java code clones and Java obfuscated code. We capture the semantics of program code using low and high level program features derived from ByteCode, the data and conditional dependencies with in the ByteCode, AST and PDG.

**Bug Detection**

Code clone detection techniques can also help in bug detection. *CP-Miner* [66] has been used to detect bugs. Higo *et al.* [40] propose an approach to efficiently detect bugs that are caused by copy-paste programming.

**Code Compacting**

Code size can be reduced by using code clone detection techniques and replacing common code using code refactoring techniques [17].

**Copyright Infringement**

Clone detection tools can easily be adapted to detect possible copyright infringement [6].
Clone Detection in Models

Model-based development can also use clone detection techniques to detect duplicated parts of models [24]. Deissenboeck et al. [25] propose an approach for automatic clone detection using large models in graph theory.

Motivation and Main Contributions

In the software engineering life cycle, code duplication not only increases maintenance costs but also causes defects which can lead to unexpected behavior. The task of maintenance is arduous usually because of inherent complexity and poor programming practices. In a large software system, it has been observed that often pairs of segments occurring in different locations are functionally identical or similar. Code obfuscation is a technique to alter the original content of the code in order to confuse reverse engineering. Malware creators use obfuscation to camouflage existing malicious code and make the task of signature based malware detection tools more challenging. Syntactic or structural signatures are weak and ineffective in detecting camouflaged codes and are overlooked easily by signature based malware detectors.

In this dissertation, four main approaches are presented to address the aforementioned problems and technical challenges. First, we take code fragments as input and find similarities among the fragments of code. Since a clone detector does not know where the repeated code fragments occur in advance, it must compare all fragments to find them. We build a model for a clone detector that uses syntactic as well as semantic information. The model uses features, which are extracted from Abstract Syntax Tree (AST), Program De-
pendency Graph (PDG), ByteCode files (BC), and ByteCode Dependency Graph (BDG).

We introduce these approaches as follows:

1. **Code Clone Detection using A Coarse and Fine-grained Hybrid Approach**
   - We use normalized blocks, followed by grouping and hashing to detect Type-I and Type-II clones.
   - We use two similarity measures to detect Type-III clones. We use the Levenshtein distance to code clone detection. Levenshtein distance is a metric for measuring the distance between two sequences. The tailored Levenshtein distance algorithm can measure distance between lines of code. We also use cosine similarity, tailored to measure angular distance between lines, represented as vectors.
   - We demonstrate that our proposed method has higher precision and F-measure than existing methods.

2. **Machine Learning Framework for Detecting Semantic Code Clones**
   - We present a simple formal model of the code clone problem and its types to better understand the issues involved.
   - We explore a new way of using features from Abstract Syntax Trees (ASTs) and Program Dependency Graphs (PDGs) to detect various types of Java code clones, including semantic clones. We believe that this attempt is the first of its kind to use features from both ASTs and PDGs to detect semantic code clones using machine learning.
   - We use state-of-the-art classification models to evaluate the effectiveness of our proposed idea.

3. **Automated Labeling Type-IV Code Clone Using Java ByteCode**
• Majority of the clone datasets used for machine learning or other approaches are incomplete in nature. They avoid labeling semantic code clones. We propose a new framework for labeling semantic code clones in Java using ByteCode similarity to label publicly available datasets, namely Suple, netbeans-javadoc, eclipse-ant, EIRC, j2sdk14.0-javax-swing, eclipse-jdtcore.

4. **Obfuscated Code Detection- An Application of Semantic Clone Detection Scheme**

• We propose an integrated framework for detecting Java code clones and obfuscated code using program or code features extracted from target pairs of codes for possible detection of clones or obfuscation.

• We use high level source code features from Abstract Syntex Tree (AST) and Program Dependency Graph (PDG) of the code.

• We explore a new way of using low level features from Java ByteCode and Byte Dependency Graph (BDG) to detect code clones and obfuscation. To the best of our knowledge this attempt is a first of its kind to use features from both Java ByteCode and BDGs to detect semantic code clones and obfuscation using machine learning.

• We use ensemble of state-of-the-art classification models to evaluate the effectiveness of our proposed idea.

**Thesis Outline**

The remainder of the thesis is organized as follows.
In Chapter II, we present literature review, which covers the background and related work on code clone detection, introducing state-of-the-art code clone detectors, along with the detection mechanisms used.

Chapter III describes how we perform a two-stage analysis, which involves coarse detection, followed by fine-grained detection. A coarse-grained analysis is used to detect Type-I and Type-II clones and the fine-grained analysis is used to detect Type-III.

In Chapter IV, we build a model based on extracting features from AST and PDG components to detect various types of Java code clones, including semantic clones using machine learning.

In Chapter V, we introduce a new dataset of clone references, which is a set of correct clones for Type-IV clones.

In Chapter VI, we propose a detection framework for detecting both Java code obfuscation and clones using an integrated machine learning scheme. BDG provide an alternative representation of semantics or meaning of a Java program. We extract novel Java ByteCode dependency graph (BDG) features to detect both Java code obfuscation and clones.

The goal is not only to compare the current status of the tools and techniques, but also to make an observation indicates that the future may lie in developing hybrid techniques. We evaluate clone detection techniques using recall, precision and F-measure metrics in addition to considering scalability, portability and clone relation to choose the right technique for a specific task.

In Chapter VII, we conclude the work in this dissertation, and propose the potential directions in the future.
Publication List

All publications and submitted papers are listed as follows.

1. Published Papers.

2. Submitted Papers.
CHAPTER II

LITERATURE REVIEW

Introduction

Recent research [6, 49, 66, 115] with large software systems [66] has detected that 22.3% of Linux code has clones. Kamiya et al. [49] have reported 29% cloned code in JDK. Baker [6] has detected clones in large systems in 13% - 20% of the source code. Baxter et al. [10] also have found that 12.7% code is cloned in a large software system. Mayrand et al. [71] have also reported that 5% - 20% code is cloned. Code clone detection can be useful for code simplification, code maintainability, plagiarism detection [68,81], copyright infringement detection, malicious software detection and detection of bug reports. Many code clone detection techniques have been proposed [86]. The focus of this chapter is to present a review of such clone detection techniques.
Categories of Detection Techniques

Detection techniques are categorized into four classes. The four classes we discuss are textual, lexical, syntactic and semantic. Syntactic approaches can be divided into tree-based and metric-based techniques, and semantic approaches can be divided into PDG-based and hybrid techniques as shown in Table 2.1. In this section, we briefly describe and compare state-of-the-art in clone detection techniques, under these classes and subclasses.

Textual Approaches

Text-based techniques compare two code fragments and declare them to be clones if the two code fragments are literally identical in terms of textual content. Text-based clone detection techniques generate fewer false positives, are easy to implement and are independent of language. Text-based clone detection techniques perform almost no transformation to the lines of source code before comparison. These techniques detect clones based on similarity in code strings and can find only Type-I clones. In this section, we discuss several well-known textual approaches or text-based techniques as shown in Table 2.2. These include Dup [6] by Baker, Duploc tool by Ducasse et al. [28], Ducasse et al. [27], Koschke et al. [56], NICAD by Roy and James [91] and SSD by Seunghak and Jeong [64].

**Dup by Baker**

Dup [6] reads source code line by line in the lexical analysis phase. Dup uses normalization, which removes comments and whitespaces and also handles identifier renaming. It hashes each line for comparison among them and extracts matches by a suffix-tree al-
Table 2.1: Techniques for clone detection.

<table>
<thead>
<tr>
<th>Approach</th>
<th>Technique</th>
<th>Tool/Author</th>
<th>Year</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>Textual</td>
<td>Text</td>
<td><em>Dup</em> 1995 [6]</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td><em>Duploc</em> 1999 [28]</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td><em>NICAD</em> 2008 [91]</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td><em>SDD</em> 2005 [64]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Lexical</td>
<td>Token</td>
<td><em>CCFinder</em> 2002 [49]</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td><em>CP-Miner</em> 2006 [66]</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td><em>Boreas</em> 2012 [115]</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td><em>FRISC</em> 2012 [77]</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td><em>CDSW</em> 2013 [76]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Syntactic</td>
<td>Tree</td>
<td><em>CloneDr</em> 1998 [10]</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td><em>Wahler et al.</em> 2004</td>
<td>[103]</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td><em>Koschke et al.</em> 2006</td>
<td>[56]</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td><em>Jiang et al.</em> 2007</td>
<td>[46]</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td><em>Hotta et al.</em> 2014</td>
<td>[44]</td>
<td></td>
</tr>
<tr>
<td>Metric</td>
<td></td>
<td><em>Mayrand et al.</em> 1996</td>
<td>[71]</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td><em>Kontogiannis et al.</em> 1996</td>
<td>[55]</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td><em>Kodhai, et al.</em> 2010</td>
<td>[52]</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td><em>Abdul-El-Hafiz et al.</em> 2012</td>
<td>[1]</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td><em>Kanika et al.</em> 2013</td>
<td>[85]</td>
<td></td>
</tr>
<tr>
<td>Semantic</td>
<td>Graph</td>
<td><em>Duplix</em> 2001 [57]</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td><em>GPLAG</em> 2006 [68]</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td><em>Higo and Kusumoto</em> 2009</td>
<td>[41]</td>
<td></td>
</tr>
<tr>
<td>Hybrid</td>
<td></td>
<td><em>ConQAT</em> 2011 [45]</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td><em>Agrawal et al.</em> 2013</td>
<td>[2]</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td><em>Funaro et al.</em> 2010</td>
<td>[34]</td>
<td></td>
</tr>
</tbody>
</table>
Table 2.2: Summary of textual approaches

<table>
<thead>
<tr>
<th>Tool</th>
<th>Transformation</th>
<th>Code Representation</th>
<th>Comparison Method</th>
<th>Complexity</th>
<th>Granularity</th>
<th>Types of Clones</th>
<th>Language Independence</th>
<th>Output Type</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Dup</strong> [6]</td>
<td>Remove whitespace and comments</td>
<td>Parameterized string matches</td>
<td>Suffix-tree based on token matching</td>
<td>$O(n + m)$ where $n$ is number of input lines and $m$ is number of matches</td>
<td>Tokens</td>
<td>Type-I</td>
<td>Needs lexer</td>
<td>Text</td>
</tr>
<tr>
<td><strong>Duploc</strong> [28]</td>
<td>Remove comments and all white space</td>
<td>Sequence of lines</td>
<td>Dynamic Pattern Matching</td>
<td>$O(n^2)$ where $n$ is number of input lines</td>
<td>Line</td>
<td>Type-I</td>
<td>Needs lexer</td>
<td>Text</td>
</tr>
<tr>
<td><strong>NICAD</strong> [91]</td>
<td>Pretty-print Methods (Segment sequences)</td>
<td>Longest Common Subsequence (LCS)</td>
<td>Longest Common Subsequence (LCS)</td>
<td>$O(n^2)$ worst case time and space where $n$ is number of lines</td>
<td>Text</td>
<td>Type-I</td>
<td>Needs parser</td>
<td>Text</td>
</tr>
<tr>
<td><strong>SDD</strong> [64]</td>
<td>No transformation and index</td>
<td>Inverted index and distance</td>
<td>N-neighbors distance</td>
<td>$O(n)$</td>
<td>Chunks of source code</td>
<td>Type-I</td>
<td>No lexer/parser needs</td>
<td>Visualization similar of code</td>
</tr>
</tbody>
</table>
algorithm. The purpose of this tool is to find maximal sections of code that are either exact
copies or near-miss clones of each other. The Dup tool can also be classified as a token-
based technique since it tokenizes each line for line-by-line matching.

To explain Dup’s technique, it is necessary to introduce the term parameterized string
(p-string), which is a string over the union of two alphabets, say $\Sigma$ and $\Pi$. It also intro-
duces the notion of parameterized match (p-match), which refers to the process in which
a p-string is transformed into another p-string by applying a renaming function. Parame-
terized matches can be detected using p-strings, which are strings that contain ordinary
characters from an alphabet $\Sigma$, and parameter characters from another finite alphabet
$\Pi$. Dup implements a p-match algorithm. The lexical analyzer produces a string contain-
ing non-parameter symbols, and zero or more parameter symbols. When sections of code
match except for the renaming of parameters, such as variables and constants, p-match
occurs. Exact match can be detected using a plain suffix tree.

The Dup algorithm encodes a p-string in the following way. The first appearance
of each parameter symbol is replaced by zero and each subsequent appearance of a pa-
rameter symbol is substituted by the distance from the previous appearance of the same
symbol. The non-parameter symbol is not changed. The Dup algorithm uses Definition
4 and Proposition 1, given below from [9, 93], to represent parameter strings in a p-suffix
tree. In Definition 4, the $f$ function, called transform, computes the $j$-th symbol value of
p-suffix($S$, $i$) in constant time from $j$ and ($j_i$-1).

A parameterized suffix tree ($P$-suffix tree) is a data structure for generalization of
suffix trees for strings. P-suffix encoding requires that a p-string $P$ and another p-string $\bar{P}$
are p-matches of each other if and only if $\text{prev}(P) = \text{prev}(\bar{P})$, where prev is the resulting
encoding of $P$. For example, when we have a p-string $T$ that has the same encoding as the p-string $P$, and $T$ and $P$ are a p-match. Therefore, $\text{prev}$ is used to test for p-matches. If $P$ is a p-string pattern and $\bar{P}$ is a p-string text, $P$ has a p-match starting at position $i$ of $T$ if and only if $\text{prev}(P)$ is a prefix of p-suffix($\bar{P},i$).

**Definition 4.** If $b$ belongs to alphabet $\sum \cup \Pi$, $f(b,j)=0$ if $b$ is a nonegative integer larger than $j-1$, and otherwise, $f(b,j)=b$ [7].

**Proposition 1.** Two p-strings $P$ and $T$ p-match when $\text{prev}(P) = \text{prev}(T)$. Also, $P < T$ when $\text{prev}(P) < \text{prev}(T)$ and $P > T$ when $\text{prev}(P) > \text{prev}(T)$ [7].

**Duploc by Ducasse et al.**

*Duploc* [28] is also a text-based clone detection technique. *Duploc* uses an algorithm that has two steps. The first step transforms source files into normalized files after eliminating noise including all whitespaces and comments. Noise elimination reduces false positives by removing common constructs. It also reduces false negatives by removing insignificant differences between code clones. The second step compares normalized files line-by-line using a simple string-matching algorithm. The hits and misses that the comparison produces are stored in a matrix and are visualized as a dotplot [27, 28, 39]. The computational complexity is $O(n^2)$ for an input of $n$ lines. Preprocessing transformed lines reduces the search space. Each line is hashed into one of a number of buckets. Every occurrence of the same hashed line value is placed in the same hash bucket. *Duploc* is able to detect a significant amount of identical code duplicates, but it is not able to identify renaming, deletion and insertion. *Duploc* does not perform lexical analysis or parsing.
The advantage of the character-based technique that *Duploc* uses is its high adaptability to diverse programming languages.

Ducasse *et al.* [27] add one more step to *Duploc*, a third step of filters. This step extracts interesting patterns in duplicated code such as gap size, which is the length of a non-repeated subsequence between a clone pair. For example, if the line sequences ‘*abcghdjgi*’ and ‘*abcfklnji*’ are compared, the gap is of length 4 because the lengths of the two non-duplicated subsequences *ghdj* and *fkln* are 4. False positives are avoided by removing noise and by filtering. The filter step uses two criteria [27]. 1) Minimum length: It is the smallest length of a sequence to be important. 2) Maximum gap size: It is the largest gap size for sequences to be obtained by copy-pasting from one another. The algorithm implements filtering in a linear amount of single matches. Ducasse’s tool uses lexical analysis to remove comments and whitespaces in the code and finds clones using a dynamic pattern matching (*DPM*) algorithm. The tool’s output is the number lines of code clone pairs. It partitions lines using a hash function for strings for faster performance. The computational complexity is $O(n^2)$, where $n$ is the input size.

Koschke *et al.* [56] prove that *Duploc* detects only Type-I clones. It cannot detect Type-II or Type-III clones or deal with modifications and insertions in copy-pasted code, called *tolerance* to modifications. *Duploc* cannot detect copy-pasted bugs [56] because detecting bugs requires semantic information and *Duploc* detects just syntactic clones.

**NICAD by Roy and James**

Roy and James [91] develop a text-based code clone detection technique called Accurate Detection of Near-miss Intentional Clones (*NICAD*). The *NICAD* tool [91] uses two
clone detection techniques: text-based and abstract syntax tree-based, to detect Type-I, Type-II and Type-III cloned code. The structures of the two approaches complement each other, overcoming the limitations of each technique alone. NICAD has three phases. 1) A parser extracts functions and performs pretty-printing that breaks different fragments of a statement into lines. 2) The second phase normalizes fragments of a statement to ignore editing differences using transformation rules. 3) The third phase checks potential clones for renaming, filtering or abstraction using dynamic clustering for simple text comparison of potential clones. The longest common subsequence (LCS) algorithm is used to compare two potential clones at a time. Therefore, each potential clone must be compared with all of the others, which makes the comparison expensive.

NICAD detects near-misses by using flexible pretty-printing. Using agile parsing [23] and the Turing eXtender Language (TXL) transformation rules [20] during parsing and pretty-printing, it can easily normalize code. By adding normalization to pretty-printing, it can detect near-miss clones with 100% similarity. After the potential clones are extracted, the LCS algorithm compares them. The NICAD tool uses percentage of unique strings (PUS) for evaluation. Equation (1) computes the percentage of unique strings for each possible clone.

If $PUS = 0\%$, the potential clones are exact clones; otherwise, if $PUS$ is more than 0\% and below a certain threshold, the potential clones are near-miss clones.

$$PUS = \frac{\text{Number of Unique Strings} \times 100}{\text{Total Number of Strings}}$$  \hspace{1cm} (2.1)
NICAD finds exact matches only when the PUS threshold is 0%. If the PUS threshold is greater than 0%, clone 1 is matched to clone 2 if and only if the size, in terms of number of lines, of the second potential clone is between \( \text{size (clone 1)} - \text{size (clone 2)} \times \text{PUS} / 100 \) and \( \text{size (clone 1)} + \text{size (clone 2)} \times \text{PUS} / 100 \). NICAD can detect exact and near-miss clones at the block level of granularity. NICAD has high precision and recall [92]. It can detect even some exact function clones that are not detected by the exact matching function used by a tree-based technique [91]. NICAD exploits the benefits of a tree-based technique by using simple text lines instead of subtree comparison to obtain good space complexity and time.

**SDD by Seunghak and Jeong**

Seunghak and Jeong [64] use a text-based code clone detection technique implemented in a tool called the Similar Data Detection (SDD), that can be used as an Eclipse plug-in. Eclipse is an integrated development environment (IDE)\(^1\). The Eclipse IDE allows the developer to extend the IDE functionality via plug-ins. SDD detects repeated code in large software systems with high performance. It also detects exact and similar code clones by using an inverted index [21] and an index data structure using a \( n \) neighbor distance algorithm [5]. The mean nearest neighbor distance is:

\[
\text{Nearest Neighbor Distance} = \frac{\sum_i \left[ \text{Min}(d_{ij}) \right]}{N} \tag{2.2}
\]

\(^1\)https://www.eclipse.org/downloads/?
where $N$ is the number of points and $Min(d_{ij})$ is the distance between each point and its nearest neighbor. $SDD$ is very powerful for detection of similar fragments of code in large systems because use of inverted index decreases $SDD$ complexity.

**Summary of Textual Approaches**

In this section, we have discussed several textual approaches for clone detection. $Dup$ [6] uses a suffix-tree algorithm to find all similar subsequences using hash values of lines, characters or tokens. The complexity of computation is $O(n)$ where $n$ is the input length of the sequence. $Duploc$ [28] uses a dynamic pattern matching algorithm to find a longest common subsequence between two sequences. $NICAD$ [91] uses the Longest Common Subsequence algorithm to compare two lines of potential clones and produces the longest sequence. The $LCS$ algorithm compares only two sequences at a time. Therefore, the number of comparisons is high because each sequence must be compared with all of the other sequences. $SDD$ [64] uses the $n$-neighbor distance algorithm to find near-miss clones. It may lead to detection of false positives.

Text-based techniques have limitations as follows [6, 28, 66]. 1) Identifier renaming cannot be handled in a line-by-line method. 2) Code fragments with line breaks are not detected as clones. 3) Adding or removing brackets can create a problem when comparing two fragments of the code where one fragment has brackets and the second fragment does not have brackets. 4) The source code cannot be transformed in text-based approaches. Some normalization can be used to improve recall without sacrificing high precision [49].
Lexical Approaches

Lexical approaches are also called token-based clone detection techniques. In such techniques, all source code lines are divided into a sequence of tokens during the lexical analysis phase of a compiler. All tokens of source code are converted back into token sequence lines. Then the token sequence lines are matched. In this section, we discuss several state-of-the-art token-based techniques as shown in Table 2.3. These include CCFinder [49] by Kamiya et al., CP-Miner [66, 91] by Zhenmin et al., Boreas [115] by Yong and Yao, FRISC [115] by Murakami et al., and CDSW [115] by Murakami et al. We choose these techniques as examples because they are among the best such techniques and can detect various types of clones with higher recall and precision than a text-based technique.

CCFinder by Kamiya et al.

Kamiya et al. [49] develop a suffix tree-matching algorithm called CCFinder. CCFinder has four phases. 1) A lexical analyzer removes all whitespaces between tokens from the token sequence. 2) Next, the token sequence is sent to the Transformation phase that uses transformation rules. It also performs parameter replacement where each identifier is replaced with a special token. 3) The Match Detection phase detects equivalent pairs as clones and also identifies classes of clones using suffix tree matching. 4) The Formatting phase converts the locations of clone pairs to line numbers in the original source files.

CCFinder applies several metrics to detect interesting clones. These metrics are given below. 1) The length of a code fragment divided by the number of tokens, or the number of
Table 2.3: Summary of lexical approaches

<table>
<thead>
<tr>
<th>Tool</th>
<th>Transformation</th>
<th>Code Representation</th>
<th>Comparison Method</th>
<th>Complexity</th>
<th>Granularity</th>
<th>Types of Clone</th>
<th>Language Independence</th>
<th>Output Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>CCFinder [49]</td>
<td>Remove whitespace, comments, and perform parameter replacement</td>
<td>Normalized sequences and parameterized tokens</td>
<td>Suffix-tree based on token matching</td>
<td>$O(n)$ where $n$ is size of source file</td>
<td>Token</td>
<td>Type-I Type-II</td>
<td>Needs lexer and transformation rules</td>
<td>Clone pairs/Clone classes</td>
</tr>
<tr>
<td>CP-Miner [66]</td>
<td>Map each statement/identifier to a number with similar statements/identifiers</td>
<td>Basic blocks</td>
<td>Frequent subsequence mining technique</td>
<td>$O(n^2)$ where $n$ is number of code lines</td>
<td>Sequence of tokens</td>
<td>Type-I Type-II</td>
<td>Needs parser</td>
<td>Clone pairs</td>
</tr>
<tr>
<td>Boreas [115]</td>
<td>Filter useless characters and extracts tokens</td>
<td>Variables matching based on other characteristics</td>
<td>Cosine similarity function</td>
<td>N/A</td>
<td>Vector</td>
<td>Type-I Type-II Type-III</td>
<td>Needs parser</td>
<td>Clusters</td>
</tr>
<tr>
<td>FRISC [77]</td>
<td>Remove whitespaces, comments, mapping from transformed sequence into original, and replace parameters</td>
<td>Hash sequence of tokens</td>
<td>Suffix array</td>
<td>N/A</td>
<td>Token sequences</td>
<td>Type-I Type-II Type-III</td>
<td>Needs lexer</td>
<td>Clone pairs/Clone classes</td>
</tr>
<tr>
<td>CDSW [76]</td>
<td>Remove whitespace, comments; map from transformed sequence into original, and parameter replacement</td>
<td>Hash values for every statement</td>
<td>Smith-Waterman Alignment</td>
<td>$O(nm)$ where $n$ is length of first token sequence and $m$ is length of second token sequence</td>
<td>Token sequences</td>
<td>Type-I Type-II Type-III</td>
<td>Needs lexer</td>
<td>Clone pairs</td>
</tr>
</tbody>
</table>
lines of the code fragment. 2) Population size of a clone class or the number of elements in a clone class. 3) Combination of the number of tokens and the number of elements in a clone class for estimating which code portion could be refactored. 4) Coverage of code clone: It is either the percentage of lines or files that include any clones. CCFinder also optimizes the sizes of programs to reduce the complexity of the token matching algorithm. It produces high recall whereas its precision is lower than that of some other techniques [51]. CCFinder accepts source files written in one programming language at a time.

The line-by-line method used by the Duploc tool [28], discussed earlier, cannot recognize or detect clones with line break relocation, when the layout of the code is changed. CCFinder performs a more suitable transformation than the line-by-line method [49]. CCFinder can also handle name changes, which the line-by-line approach cannot handle. However, CCFinder or a token-based technique takes more CPU time and more memory than line-by-line comparison [28, 49, 66]. CCFinder uses a suffix tree algorithm, and so it cannot handle statement insertions and deletions in code clones [66].

**CP-Miner by Li et al.**

Li et al. [66] use a token-based technique to detect code clones and clones related to bugs in large software systems. Their system, CP-Miner, searches for copy-pasted code blocks using frequent subsequence mining [111]. CP-Miner implements two functions.

1. **Detecting copy-pasted code fragments.** CP-Miner converts the problem into a frequent subsequence mining problem by parsing source code to build a database containing a collection of sequences. It then implements an enhanced version of the CloSpan
algorithm [111], which is used to help satisfy gap constraints in frequent subsequences. Each similar statement is mapped to the same token.

(ii) **Finding copy-paste related bugs.** Frequently, programmers forget to rename identifiers after copy-pasting. Unchanged identifiers are detected by a compiler and reported as “errors”. These errors become unobserved bugs that can be very hard to detect by a detector. Therefore, **CP-Miner** uses an *UnchangedRatio* threshold to detect bugs.

\[
UnRenamed_{IDRate} = \frac{NumUnchanged_{ID}}{TotalUnchanged_{ID}}
\]  

(2.3)

where *UnRenamed_{IDRate}* is the percentage of unchanged identifiers, *NumUnchanged_{ID}* is the number of unchanged identifiers and *TotalUnchanged_{ID}* is the total number of identifiers in a given copy-pasted fragment. The value of *UnRenamed_{IDRate}* can be any value in the range 0 and 1. If *UnRenamed_{IDRate}* is 0, it means that all occurrences of identifiers have been changed, and if *UnRenamed_{IDRate}* is 1, it means that all occurrences of the identifier remain unchanged.

**CP-Miner** can only detect forgot-to-change bugs. This means that if the programmer has forgotten to modify or insert some extraneous statements to the new copy-pasted segment, **CP-Miner** would not detect the bug because the changed code fragments are now too different from the others [66]. This approach can detect similar sequences of tokenized statements and avert redundant comparisons, and as a result, **CP-Miner** detects code clones efficiently, even in millions of code lines.

**CP-Miner** overcomes some limitations of **CCFinder** and detects more copy-pasted segments than **CCFinder** does. However, **CCFinder** does not detect code clones that are
related to bugs as *CP-Miner* does because *CP-Miner* uses an unchanged ratio threshold. *CCFinder* does not completely filter false positives and it detects many tiny cloned code blocks which seem to be predominantly false positives. Because *CP-Miner* handles statement insertions and modifications, *CP-Miner* can detect 17-52% more code clones than *CCFinder*. Unfortunately, the frequent subsequence mining algorithm that *CCFinder* uses has two limitations because it divides a long sequence into sub-sequences. First, some frequent subsequences of two or more statement blocks may be lost. Second, it is hard to choose the size of short sequences because if the size is too short, the information may be lost; if the size is too long, the mining algorithm may be very slow [66].

**Boreas by Yang and Yao.**

Yang and Yao [115] use a token-based approach called *Boreas* to detect clones. *Boreas* uses a novel counting method to obtain characteristic matrices that identify program segments effectively. *Boreas* matches variables instead of matching sequences or
structures. It uses three terms 1) The Count Environment, 2) The Count Vector, and 3) The Count Matrix. These are discussed below.

The computation of the Count Environment \((CE)\) is divided into three stages. The first stage is Naïve Counting, which counts the number variables used and defined in the environments. The second stage is In-statement counting, which counts the number of regular variables as well as the variables used as if-predicates and array subscripts, and variables that are defined by expressions with constants. The third stage is Inter-statement Counting, which counts variables used inside a first-level loop, second level loop or third level loop. 2) Count Vector \((CV)\), which is produced using \(m\) \((m\)-dimensional Count Vector) \(CEs\). The \(i\)-th dimension in the \(CV\) is the number of variables in the \(i\)-th \(CE\). \(CV\) is also called a characteristic vector. 3) Counting Matrix \((CM)\), which contains all \(n\), which is number of variables, \((n\)-variables) \(CVs\) in code fragments and is an \(n \times m\) Count Matrix.

\textit{Boreas} uses the cosine of the angle between the two vectors to compare similarity:

\[
Sim(v_1, v_2) = \cos(\alpha) = \frac{\sum_{i=1}^{n} v_{1i} \times v_{2i}}{\sqrt{\sum_{i=1}^{n} v_{1i}^2 \times \sum_{i=1}^{n} v_{2i}^2}} \tag{2.4}
\]

where \(Sim\) is the cosine similarity between two vectors \(v_1\) and \(v_2\), and \(\alpha\) is the angle between them. The similarity between two fragments is measured by an improved proportional similarity function. This function compares the \(CVs\) of keywords and punctuations marks. \textit{PropSimilarity} is proportional similarity between \(C_1\) and \(C_2\), which are two occurrences counts:

\[
PropSimilarity = \frac{1}{(C_1 + 1)} + \frac{C_2}{(C_1 + 1)}. \tag{2.5}
\]
The function prevents incorrect zero similarity. Boreas is not able to detect code clones of Type-III. Agrawal et al. [2] extend Boreas to detect clones by using a token-based approach to match clones with one variable or a keyword and easily detect Type-I and Type-II clones; they use a textual approach to detect Type-III clones. Since Agrawal et al.’s approach combines two approaches, it is a hybrid approach.

**FRISC by Murakami et al.**

Murakami et al. [77] develop a token-based technique called FRISC which transforms every repeated instruction into a special form and uses a suffix array algorithm to detect clones. FRISC has five steps. 1) Performing lexical analysis and normalization, which transforms source files into token sequences and replaces every identifier by a special token. 2) Generating statement hash, which generates a hash value for every statement between “;”, “{”, and “}” with every token included in a statement. 3) Folding repeated instructions, which identifies every repeated subsequence and divides into the first repeated element and its subsequent repeated elements. The repeated subsequences are removed and their numbers of tokens are added to their first repeated subsequence of elements. 4) Detecting identical hash subsequences, which detects identical subsequences from the folded hash sequences. If the sum of the numbers of tokens is smaller than the minimum token length, they are not considered clones. 5) Mapping identical subsequences to the original source code, which converts clone pairs to original source code.

FRISC supports Java and C. The authors performed experiments with eight target software systems, and found that the precision with folded repeated instructions was higher than the precision without by 29.8%, but the recall decreased by 2.9%. FRISC has higher
recall and precision than CCFinder but the precision is lower than CloneDr [10] and CLAN [72].

**CDSW by Murakami et al.**

Murakami et al. [76] develop another token-based technique, called CDSW, which detects Type-III clones (gapped clones) using the Smith-Waterman algorithm [99]. It eliminates the limitations of AST-based and PDG-based techniques, which require much time to transform source code into ASTs or PDGs, and compare among them. CDSW has five steps. 1) Performing lexical analysis and normalization, which is the same as the first step of FRISC. 2) Calculating hash values for every statement, which is the same as FRISC. 3) Identifying similar hash sequences, which identifies similar hash sequences using the Smith-Waterman Algorithm. 4) Identifying gapped tokens using the Longest Common Subsequence (LCS) algorithm to identify every sequence gap. 5) Mapping identical subsequences to the source code, which converts clone pairs to the original source code. It is also performs the same fifth step as FRISC.

Since Bellon’s references, which are built by manually confirming a set of candidates to be clones or clone pairs that are judged as correct [11], do not contain gapped fragments, Murakami et al. enhance the set of clone references by adding information about gapped lines. Murakami et al. calculate recall, precision, and f-measure using Bellon’s [11] and their own clone references resulting in improved Recall, Precision, and F-measure. Recall increased by 4.1%, Precision increased by 3.7%, and F-measure increased by 3.8% in the best case. Recall increased by 0.49%, Precision increased by 0.42% and F-measure increased by 0.43% in the worst case [76]. The results are different because CDSW replaces
all variables and identifiers with special tokens that ignore their types. Because CDSW does not normalize all variables and identifiers, it cannot detect clones that have different variable names [76].

**SourcererCC by Sajnani et al.**

SourcererCC [95] is a token-based syntactic and semantic clone detection method that uses an optimized partial index of tokens and filtering heuristics to achieve large-scale detection.

**Summary of Lexical Approaches**

The suffix-tree based token matching algorithm used by *CCFinder* finds all similar subsequences in a transformed token sequence. *CCFinder* cannot detect statement insertions and deletions in copy-pasted code. It does not completely eliminate false positives. The frequent subsequence mining technique used by *CP-Miner* discovers frequent subsequences in a sequence database. A frequent subsequence mining technique avoids unnecessary comparisons, which makes *CP-Miner* efficient. *CP-Miner* detects 17%-52% more code clones than *CCFinder*. A limitation of a frequent subsequence mining algorithm is that a sequence database is needed. *Boreas* works fast by using two functions: cosine similarity and proportional similarity. *FRISC* detects more false positives than the other tools but misses some clone references [77]. *CDSW*’s accuracy is based on the *match*, *mismatch* and *gap* parameters. If these parameters are changed, the results are different.

Token-based techniques have limitations as follows. 1) Token-based techniques depend on the order of program lines. If the statement order is modified in duplicated code,
the duplicated code will not be detected. 2) These techniques cannot detect code clones with swapped lines or even added or removed tokens because the clone detection is focused on tokens. 3) Token-based techniques are more expensive in time and space complexity than text-based techniques because a source line contains several tokens.

**Syntactical Approaches**

Syntactical approaches are categorized into two kinds of techniques. The two categories are tree-based techniques and metric-based techniques. A list of syntactical techniques found in the literature is shown in Table 2.4. In this section, we discuss several common tree-based and metric-based techniques. For the purpose of this study, we choose **CloneDR** [10] by Baxter et al., Wahler [103], Koschke [56], Jiang [46], Mayrand et al. [71], Kontogiannis et al. [55], Kodhai, et al. [52], Abdul-El-Hafiz [1] and Kanika et al. [85].

**Tree-based Clone Detection Techniques**

In these techniques, the source code is parsed into an abstract syntax tree (AST) using a parser and the sub-trees are compared to find cloned code using tree-matching algorithms.

**CloneDr by Baxter et al.** Baxter et al. [10] use a tree-based code clone detection technique implemented in a tool called **CloneDr**. It can detect exact clones, near-miss clones and refactored code using an AST. After the source code is parsed into an AST, it finds clones by applying three main algorithms. The first algorithm detects sub-tree clones, the second algorithm detects variable-size sequences of sub-tree clones such as sequences of declarations or statements, and the third algorithm finds more complex near-miss clones by
Table 2.4: Summary of syntactical approaches

<table>
<thead>
<tr>
<th>Author/Tool</th>
<th>Transformation</th>
<th>Code Representation</th>
<th>Comparison Method</th>
<th>Complexity</th>
<th>Granularity/Types of Clone</th>
<th>Language Independence</th>
<th>Output Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>CloneDr [10]</td>
<td>Parse to AST</td>
<td>AST</td>
<td>Tree matching technique</td>
<td>$O(n)$ where $n$ is number of AST nodes</td>
<td>AST node</td>
<td>Type-I, Type-II</td>
<td>Needs parser</td>
</tr>
<tr>
<td>Wahler [103]</td>
<td>Parse to AST</td>
<td>AST</td>
<td>Frequent itemset</td>
<td>N/A</td>
<td>Line</td>
<td>Type-I, Type-II</td>
<td>Needs parser</td>
</tr>
<tr>
<td>Koschke [56]</td>
<td>Parse to AST</td>
<td>AST</td>
<td>Simple string suffix tree algorithm</td>
<td>$O(n)$ where $n$ is number of input nodes</td>
<td>Tokens</td>
<td>Type-I, Type-II, Type-III</td>
<td>Needs parser</td>
</tr>
<tr>
<td>Jung et al. [46]</td>
<td>Parse to parse tree then to a set of vectors.</td>
<td>Vectors</td>
<td>Locality-sensitive hashing Tree-Matching algorithm (LSH)</td>
<td>$O(\ell(T_1)\ell(T_2)d_1d_2)$, where $\ell(T_i)$ is the size of $T_i$ and $d_i$ is the minimum of the depth of $T_i$ and the number of leaves of $T_i$</td>
<td>Vectors</td>
<td>Type-I, Type-II, Type-III</td>
<td>Needs parser</td>
</tr>
<tr>
<td>Hotta et al. [44]</td>
<td>Parse source code to extract blocks using JDT</td>
<td>Hashed blocks</td>
<td>Group blocks based on hash values</td>
<td>N/A</td>
<td>Blocks</td>
<td>Type-I, Type-II</td>
<td>Needs parser</td>
</tr>
<tr>
<td>Mayrand et al. [71]</td>
<td>Parse to AST then (IRL)</td>
<td>Metrics</td>
<td>21 function metrics</td>
<td>Polynomial complexity</td>
<td>Metrics for each function</td>
<td>Type-I, Type-II, Type-III</td>
<td>Needs Dartix tool</td>
</tr>
<tr>
<td>Kontogiannis et al. [55]</td>
<td>Transform to feature vectors</td>
<td>Feature vectors</td>
<td>Use numerical comparisons of metric values and dynamic programming (DP) using minimum edit distance</td>
<td>$O(n^2)$ for Naïve approach and $O(</td>
<td>T</td>
<td>)$ for DP-model</td>
<td>Metrics of a begin-end block</td>
</tr>
<tr>
<td>Kodhai et al. [52]</td>
<td>Remove whitespaces, comments, mapping and pre-process statements</td>
<td>Metrics</td>
<td>The string matching/textual comparison</td>
<td>N/A</td>
<td>Functional</td>
<td>Type-I, Type-II</td>
<td>Needs parser</td>
</tr>
<tr>
<td>Abdul-El-Hafiz et al. [1]</td>
<td>Preprocess and extract Metrics</td>
<td>Metrics</td>
<td>Data mining clustering algorithm and fractal clustering</td>
<td>$O(M^2\log(M))$ where $M$ is the size of data set</td>
<td>Functional</td>
<td>Type-I, Type-II, can be Type-III</td>
<td>Language independent</td>
</tr>
<tr>
<td>Kanika et al. [85]</td>
<td>Calculate metrics of Java programs</td>
<td>Metrics</td>
<td>Use 3-phase comparison algorithm: Adaptation, Computation and Measurement Phases</td>
<td>N/A</td>
<td>Metrics of Java Byte-Code</td>
<td>Type-I, Type-II, Type-III</td>
<td>Needs compiler</td>
</tr>
</tbody>
</table>
generalizing other clone combinations. The method splits sub-trees using a hash function and then compares sub-trees in the same bucket. The first algorithm finds sub-tree clones and compares each sub-tree with other sub-trees. Near-miss clones that cannot be detected by comparing sub-trees can be found using similarity computation:

\[
Similarity = \frac{2SN}{2SN + LEFT + RIGHT}
\]  

(2.6)

where \(SN\) is the number of shared nodes, \(LEFT\) is the number of nodes in sub-tree1 and \(RIGHT\) is the number of nodes in sub-tree2. The second algorithm finds clone sequences in \(ASTs\). It compares each pair of sub-trees and looks for maximum length sequences. The third algorithm finds complex near-miss clones. It abstracts each pair of clones after all clones are detected.

\textit{CloneDr} cannot detect semantic clones. Text-based techniques do not deal with modifications such as renaming of identifiers since there is no lexical information. Tree-based techniques may produce false positives since two fragments of the sub-tree may not be duplicated. Because a tree-based method hashes subtrees, it cannot detect duplicated code which has modifications.

\textbf{Wahler et al.} Wahler et al. [103] detect clones which are represented as an abstract syntax tree (\(AST\)) in XML by applying frequent itemset mining. Frequent itemset mining is a data mining technique that looks for sequences of actions or events that occur frequently. An instance is called a \textit{transaction}, each of which has a number of features called \textit{items}. This tool uses frequent itemsets to identify features in large amounts of data using the Apriori
algorithm [38]. For each itemset, they compute its support count, which is the frequency of occurrence of an itemset or the number of transactions in which it appears:

$$\sigma(I) = \frac{\left| \{ T \in D \mid I \subseteq T \} \right|}{|D|} \geq \sigma$$

(2.7)

where $T$ is a transaction, $I$ is an itemset which is a subset of the transaction $T$, and $D$ is a database. If an itemset’s frequency is more than a certain given support count $\sigma$, it is called a frequent itemset.

There are two steps to find frequent itemsets. The first step is the join step. The first step finds $L_k$, which are frequent itemsets of size $k$. A set of candidate $k$-itemsets is generated by combining $L_{k-1}$ with itself. The second step is the prune step, which finds frequent $k$-itemsets from $C_k$. This process is repeated until no more frequent $k$-itemsets are found. In this approach, the statements of a program become items in the database $D$.

Clones are a sequence of source code statements that occur more than once. Therefore, the support count is $\sigma = \frac{2}{|D|}$. Let there be statements $b_1...b_k$ in a program. The join step combines two frequent $(k-1)$-itemsets of the form $I_1 = b_1...b_k$, $I_2 = b_2...b_{k-1}$.

Koschke et al. Koschke et al. [56] also detect code clones using an abstract syntax tree (AST). Their method finds syntactic clones by pre-order traversal, applies suffix tree detection to find full subtree copies, and decomposes the resulting Type-I and Type-II token sequences. This approach does not allow structural parameters. It can find Type-I and Type-II clones in linear time and space. AST-based detection can be used to find syntactic clones with more effort than Token-based suffix trees and with low precision. AST-based detec-
tion also scales worse than Token-based detection. Token-based suffix tree clone detectors can be adapted to a new language in a short time whereas using AST needs a full abstract syntax tree and sub-tree comparison method. Using abstract syntax suffix trees [56] detects clones in less time.

**Deckard by Jiang et al.** Jiang et al. [46] also use the tree-based technique and compute certain characteristic vectors to capture structural information about ASTs in Euclidean space. Locality Sensitive Hashing (LSH) [22] is a technique for clustering similar items using the Euclidean distance metric. The Jiang et al. tool is called Deckard. Deckard’s phases include the following. 1) A parser uses a formal syntactic grammar and transforms source files into parse trees. 2) The parse trees are used to produce a set of vectors that capture structural information about the trees. 3) The vectors are clustered using the Locality Sensitive Hashing algorithm (LSH) that helps find a query vector’s near-neighbors. Finally, post-processing is used to generate clone reports. Deckard detects re-ordered statements and non-contiguous clones.

*Deckard* [46] is language independent with lower speed than the *Boreas* tool, discussed in Subsection, because of less set-up time and less comparison time [66, 115]. *Deckard* also requires constructing ASTs, which requires more time.

**Hotta et al.** Hotta et al. [44] compare and evaluate methods for detection of coarse-grained and fine-grained unit-level clones. They use a coarse-grained detector that detects block-level clones from given source files. Their approach has four steps. 1) Lexical and syntactic analysis to detect all blocks from the given source files such as classes, methods and block
Figure 2.2: Example of coarse-grained clone detection. [44]

1- `public class C1
2- {
3-   int result = 1;
4-   int result1 = 7;
5-   private int method(int x)
6-   {
7-     for (int i = 1; i < x; i++)
8-       result = result + i;
9-   }
10- for (int j = 1; j < y; j++)
11-     result = result + j;
12- }
13- }
14- private int method1(int x)
15- {
16-   for (int i = 1; i <= x; i++)
17-     result = result * i;
18-     result1 = result - i;
19- }
20- }
21- }

//Block1
`public class S
{
  int S = 5;
  int S = 5;
//Block2
private int $int S$
{
//Block4
  for (int S = S; S < S; S++)
  S = S + S;
//Block5
  for (int S = S; S < S; S++)
  S = S + S;
}
//Block3
private int $int S$
{
  for (int S = S; S <= S; S++)
  S = S + S;
  S = S + S;
}

Hash Value Block
100 Block1
200 Block2
300 Block4
400 Block6

Block4 and Block5 are a clone pair

statements. 2) Normalization of every block detected in the previous step. This step detects Type-I and Type-II clones. 3) Hashing every block using the `hashCode()` function `java.lang.String`. 4) Grouping blocks based on their hash values. If two normalized blocks have the same hash value, they are considered equal to each other as in Figure 2.2. The detection approach has high accuracy, but Hotta et al.’s method, which is coarse-grained does not have high recall compared to fine-grained detectors, does not tackle gapped code clones, and detects fewer clones. Their approach is much faster than a fine-grained approach, since the authors use hash values of texts of blocks. However, using a coarse-grained approach alone is not enough because it does not have more detailed information about the clones. A fine-grained approach must be used as a second stage after a coarse-grained approach.
Metric-based clone detection techniques.

In metric-based clone detection techniques, a number of metrics are computed for each fragment of code to find similar fragments by comparing metric vectors instead of comparing code or ASTs directly. Seven software metrics have been used by different authors [52, 71, 72].

1. Number of declaration statements,
2. Number of loop statements,
3. Number of executable statements,
4. Number of conditional statements,
5. Number of return statements,
6. Number of function calls, and
7. Number of parameters.

All of these metrics are computed and their values are stored in a database [52]. Pairs of similar methods are also detected by comparison of the metric values, which are stored in the same database.

Mayrand et al. Mayrand et al. [71] compute metrics from names, layouts, expressions and control flows of functions. If two functions’ metrics are similar, the two functions are considered to be clones. Their work identifies similar functions but not similar fragments of code. In reality, similar fragments of code occur more frequently than similar functions.

First, source code is parsed to an abstract syntax tree (AST). Next, the AST is translated into an Intermediate Representation Language (IRL) to detect each function. This tool
reports as clone pair two function blocks with similar metrics values. Patenaude et al. [79] extend Mayrand’s tool to find Java clones using a similar metric-based algorithm.

**Kontogiannis et al.** Kontogiannis et al. [54] propose a way to measure similarity between two pieces of source code using an abstract pattern matching tool. Markov models are used to calculate dissimilarity between an abstract description and a code fragment. Later, they propose two additional methods for detecting clones [55]. The first method performs numerical comparison of the metric values that categorize a code fragment to begin-end blocks. The second approach uses dynamic programming to compute and report begin-end blocks using minimum edit distance. This approach only reports similarity measures and the user must go through block pairs and decide whether or not they are actual clones.

**Kodhai et al.** Kodhai et al. [52] combine a metric-based approach with a text-based approach to detect functional clones in C source code. The process of clone detection has five phases. 1) The Input and Pre-processing step parses files to remove pre-processor statements, comments and whitespaces. The source code is rebuilt to a standard form for easy detection of similarity of the cloned fragments. 2) Template conversion is used in the textual comparison of potential clones. It renames data types, variables and function names. 3) Method identification identifies each method and extracts them. 4) Metric Computation. 5) Type-I and Type-II clone detection. The text-based approach finds clones with high accuracy and reliability, but the metric-based approach can reduce the high complexity of the text-based approach by using computed metrics values. The limitation of this method is that it just detects Type-I and Type-II clones, with high time complexity.
Abdul-El-Hafiz et al. Abdul-El-Hafiz et al. [1] use a metric based data mining approach. They use a fractal clustering algorithm. This technique uses four processes. 1) Preprocessing the input source file. 2) Extracting all fragments to analyze and related metrics. 3) Partitioning the set of fragments into a small number of clusters of three types using fractal clustering. Primary clusters cover Type-I and Type-II clones, Intermediate clusters cover Type-III clones, and a singleton cluster has only one function that is not a clone of any other functions. 4) Post-processing, which extracts clone classes from the primary cluster. This technique uses eight metrics which detect each type of function.

MCD Finder by Kanika et al. Kanika et al. [85] use a metric-based approach to develop the MCD Finder tool for Java. Their tool performs a metric calculation on the Java Byte-Code instead of directly on the source code. This approach consists of three phases. 1) The Java source code is compiled to make it adaptable to requirement of the tool. 2) The computation phase computes metrics that help detect potential clones. This approach uses 9 metrics [85] for each function.

1. Number of calls from a function,
2. Number of statements in a function,
3. Number of parameters passed to a function,
4. Number of conditional statements in a function,
5. Number of non-local variables inside a function,
6. Total number of variables inside a function,
7. Number of public variables inside a function,
8. Number of private variables inside a function, and
9. Number of protected variables inside a function.

**Wang et al.** Wang *et al.* [105] propose a syntactic clone detection method using a Bayesian Network framework with a set of features such as history, code and destination features. Yang *et al.* [113] propose a user feedback based learning model, FICA to detect Type-I, II, and III clones. Wang *et al.* extract textual features to use with the machine learning model. Recently, a language model coupled with deep learning has been used to detect code clones [108]. They use a greedy approach for AST generation and extract features to train the model. The calculated metrics are stored in a database and mapped onto Excel sheets. The measurement phase performs a comparison on the similarity of the metric values.

**Summary of Syntactical Approaches**

In Table 2.4, we provide a summary of syntactical techniques considering several properties. The first kind of syntactical approach is the tree-based technique. One such system, *CloneDr*, finds sub-tree clones with limitations as follows. 1) It has difficulty performing near-miss clone detection, but comparing trees for similarity solves it to some extent. 2) Scaling up becomes hard when the software system is large and the number of comparisons becomes very large. Splitting the comparison sub-trees with hash values solves this problem. The parser also parses a full tree. Wahler *et al.*’s approach detects Type-I and Type-II clones only. The clones are detected with a very low recall. *Deckard* detects significantly more clones and is much more scalable than Wahler *et al.*’s technique because
*Deckard* uses characteristic vectors and efficient vector clustering techniques. Koschke *et al.* show that suffix-tree clone detection scales very well since a suffix tree finds clones in large systems and reduces the number of subtree comparisons.

The second kind of syntactical approach is the metric-based technique. Mayrand *et al.*’s approach does not detect duplicated code at different granularities. Kontogiannis *et al.*’s approach works only at block level, it cannot detect clone fragments that are smaller than a block, and it does not effectively deal with renamed variables or work with non-contiguous code clone. Kodhai *et al.*’s approach detects Type-I and Type-II clones only. The limitation of Abdul-El-Hafiz *et al.*’s technique is that Type-IV clones cannot be detected. The *MCD Finder* is efficient in detecting semantic clones because ByteCode is platform independent whereas *CCFinder* cannot detect semantic clones. However, even *MCD Finder* tool cannot detect all clones.

Syntactical techniques have limitations as follows. 1) Tree-based techniques do not handle identifiers and literal values for detecting clone in ASTs. 2) Tree-based techniques ignore identifier information. Therefore, they cannot detect reordered statement clones. 3) Metric-based techniques need a parser or *PDG* generator to obtain metrics values. 4) Two code fragments with the same metric values may not be similar code fragments based on metrics alone.

**Semantic Approaches**

A semantic approach, which detects two fragments of code that perform the same computation but have differently structured code, uses static program analysis to obtain more accurate information similarity than syntactic similarity. Semantic approaches are
categorized into two kinds of techniques. The two kinds are Graph-based techniques and Hybrid techniques. Several semantic techniques from the literature are shown in Table 2.5. In this section, we discuss Komondoor and Horwitz [53], Duplix [57] by Krinke, GPLAG [68] by Liu et al., Higo and Kusumoto [41], Hummel et al. [45], Funaro et al. [34], and Agrawal et al. [2].

**Graph-based Clone Detection Techniques**

A graph-based clone detection technique uses a graph to represent the data and control flow of a program. One can build a program Dependency Graph (PDG) as defined in Section . Because a PDG includes both control flow and data flow information as given in Definitions 5 and 6, respectively, one can detect semantic clones using PDG [98]. Clones can be detected as isomorphic subgraphs [53]. In PDG edges represent the data and control dependencies between vertices which repeat lines of code, in PDGs.

**Definition 5. (Control Dependency Edge).** There is a control dependency edge from a vertex to a second program vertex in a Program Dependency Graph if the truth of the condition controls whether the second vertex will be executed [68].

**Definition 6. (Data Dependency Edge).** There is a data dependency edge from program vertex var\(_1\) to var\(_2\) if there is some variable such that:

- var\(_1\) may be assigned a value, either directly or indirectly through pointers.

- var\(_2\) may use the value of the variable, either directly or indirectly through pointers.

- There is an execution path in the program from the code corresponding to var\(_1\) to the code corresponding to var\(_2\) along which there is no assignment to variable [68].
<table>
<thead>
<tr>
<th>Author/Tool</th>
<th>Transformation</th>
<th>Code Representation</th>
<th>Comparison Method</th>
<th>Complexity</th>
<th>Granularity/Types of Clone</th>
<th>Language Independence</th>
<th>Output Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Komondoor and Horwitz [53]</td>
<td>PDGs</td>
<td>PDGs using CodeSurfer</td>
<td>PDGs Isomorphic PDG subgraph matching using backward slicing</td>
<td>N/A</td>
<td>PDG node Type-III Type-IV</td>
<td>Needs tool for converting source code to PDGs Clone pairs and Clone Classes</td>
<td></td>
</tr>
<tr>
<td>Duplix [57]</td>
<td>To PDGs</td>
<td>PDGs</td>
<td>K-length patch algorithm</td>
<td>Non-polynomial complexity</td>
<td>PDG Subgraphs Type-I Type-II Type-IV</td>
<td>Needs tool for converting source code to PDGs Clone Classes</td>
<td></td>
</tr>
<tr>
<td>GPLAG [68]</td>
<td>PDGs</td>
<td>PDGs using CodeSurfer</td>
<td>Isomorphic PDG subgraph matching algorithm</td>
<td>NP-Complete</td>
<td>PDG Node Type-I Type-II Type-III</td>
<td>Needs tool for converting source code to PDGs Plagiarized pair of programs</td>
<td></td>
</tr>
<tr>
<td>Higo and Kusumoto [41]</td>
<td>To PDGs</td>
<td>PDGs</td>
<td>Code Clone Detection Module</td>
<td>N/A</td>
<td>Edges Type-III</td>
<td>Needs tool for converting source code to PDGs Clone Pairs Files</td>
<td></td>
</tr>
<tr>
<td>ConQAT [45]</td>
<td>Splits source code into tokens and removes comments and variable names. Then normalized tokens are grouped into statements</td>
<td>Tokens</td>
<td>Suffix-tree-based algorithm then using Index-based clone detection algorithm</td>
<td>$O(</td>
<td>f</td>
<td>\log N)$ where $f$ is the number of statements and $N$ is the number of stored tuples</td>
<td>Substrings Type-I Type-II</td>
</tr>
<tr>
<td>Funaro et al. [34]</td>
<td>Parsed to AST, then Serialized AST</td>
<td>AST</td>
<td>Textual comparison</td>
<td>N/A</td>
<td>Specific parts of AST Type-I Type-II Type-III</td>
<td>Needs parser String clones</td>
<td></td>
</tr>
<tr>
<td>Agrawal et al. [2]</td>
<td>To tokens</td>
<td>Tokens</td>
<td>Line-by-line textual comparison</td>
<td>N/A</td>
<td>Indexed tokens Type-I Type-II Type-III</td>
<td>Needs lexer Text clones</td>
<td></td>
</tr>
</tbody>
</table>
Komondoor and Horwitz Komondoor and Horwitz [53] use program slicing [107] to find isomorphic PDG subgraphs and code clones. As mentioned earlier, nodes in a PDG represent statements and predicates, and edges represent data and control dependences. The slicing clone detection algorithm performs three steps. 1) Find pairs of clones by partitioning all PDG nodes into equivalence classes, where any two nodes in the same class are matching node [53]. 2) Remove subsumed clones. A clone pair subsumes another clone pair if and only if each element of the clone pair is a subset of another element from another clone pair. So, subsumed clone pairs need to be removed. 3) Combine pairs of clones into larger groups using transitive closure.

Duplix by Krinke Krinke [57] finds maximally similar PDG subgraphs with high precision and recall. Krinke’s approach is similar to the Komondoor and Horwitz approach [53], which starts from every pair of matching nodes and uses sub-graphs that are not maximal and are just subtrees unlike the ones in [57]. The PDG used by Krinke is similar to AST and the traditional PDG. Thus, the PDG contains vertices and edges that represent components of expressions. It also contains immediate (control) dependency edges. The value dependency edges represent the data flow between expression components. Another edge, the reference dependency edge, represents the assignments of values to variables.

GPLAG by Liu et al. Liu et al. [68] develop an approach to detect software plagiarism by mining PDGs. Their tool is called GPLAG. Previous plagiarism detection tools were only partially sufficient for academic use in finding plagiarized programs in programming classes. These tools were based on program token methods such as JPlag [81] and were
unable to detect disguised plagiarized code well. Plagiarism disguises may include the following [68]. 1) Format alteration such as inserting and removing blanks or comments. 2) Variable renaming where variables names may be changed without affecting program correctness. 3. Statement reordering, when some statements may be reordered without affecting the results. 4) Control replacement such as a for loop can be substituted by a while loop and vice versa. For example, \(\text{for } (\text{int } i=0; i<10; i++) \{a=b-c;\} \)  block can be replaced by \(\text{while } (i<10) \{a=b-c; i++; \}\). 5) Code Insertion, where additional code may be inserted to disguise plagiarism without affecting the results.

**Scorpio by Higo and Kusumoto** Higo and Kusumoto [41] propose a PDG-based incremental two-way slicing approach to detect clones, called Scorpio. Scorpio has two processes: 1) Analysis processing: The inputs are the source files to the analysis module and the output is a database. PDGs are generated from the algorithms of source files. Then, all the edges of the PDGs are extracted and stored in a database. 2) Detection processing: The inputs are source files and the database, and the output is a set of clone pairs. First, a user provides file names and the edges are retrieved for the given files from the database. Finally, the clone pairs are detected by the detection model. This approach detects non-contiguous clones while other existing incremental detection approaches cannot detect non-contiguous clones. The approach also has faster speed compared to other existing PDG based clone detection approaches.
Hybrid Clone Detection Techniques

A hybrid clone detection technique uses a combination of two or more techniques. A hybrid approach can overcome problems encountered by individual tools or techniques.

**ConQAT.** Hummel et al. [45] use a hybrid and incremental index based technique to detect clones and implement a tool called ConQAT. Code clones are detected in three phases. 1) Preprocessing, which divides the source code into tokens, normalizes the tokens to remove comments or renamed variables. All normalized tokens are collected into statements. 2) Detection, which looks for identical sub-strings. 3) Post-processing, which creates code cloning information looking up all clones for a single file using a clone index. Statements are hashed using MD5 hashing [88]. Two entries with the same hash sequence are a clone pair. The approach extracts all clones for a single file from the index and reports maximal clones.

**Funaro et al.** Funaro et al. [34] propose a hybrid technique that combines a syntactic approach using an abstract syntax tree to identify potential clones with a textual approach to avoid false positives. The algorithm has four phases: 1) Building a forest of ASTs. 2) Serializing the forest and encoding into a string representation with an inverse mapping function. 3) Seeking serialized clones. 4) Reconstructing clones.

**Agrawal et al.** Agrawal et al. [2] present a hybrid technique that combines token-based and textual approaches to find code cloning to extend Boreas [115], which cannot detect Type-III code clones. The token approach can easily detect Type-I and Type-II code clones.
The textual approach can detect Type-III code clones that are hard to detect with the token approach. The technique has three phases. 1) The pre-processing phase removes comments, whitespaces and blank lines. Declarations of variables in a single line are combined to make it easy for the tool to find the number of variables declared in the program. 2) The transformation phase breaks the source code into tokens and detects Type-I and Type-II code clones. 3) The match detection phase finds code clones using a matrix representation and then replaces each token with an index value. Then a textual approach looks for the same string values line-by-line. In this phase, Type-III code clones are detected. 4) The filtering phase removes false positives.

Summary of Semantic Approaches

In Table 2.5, we provide a summary of semantic techniques. The first kind of semantic approaches includes PDG-based techniques. The approach by Komondoor and Horwitz needs a tool to generate the PDG subgraph. But, the major benefit of the Komondoor and Horwitz tool is that it can detect gapped clones. The Komondoor and Horwitz tool and Duplix detect semantically robust code clones using PDG for procedure extraction, which is a program transformation that can be used to make programs easier to understand and maintain. Duplix cannot be applied to large systems and is very slow. Tools that do not use PDG can be effectively confused by statement reordering, replacing and code insertion. Since PDG is robust to the disguises that confuse other tools, GPLAG is more effective and efficient than these tools. GPLAG has a limitation that the computational complexity increases exponentially with the size of the software code. Scorpio by Higo and Kusumoto
detects non-contiguous clones while other incremental detection approaches cannot do so. It also has faster speed than other PDG based clone detection approaches.

The second kind of semantic approaches is represented by hybrid techniques. Hummel et al. use an approach similar to ConQAT, but it is different because Hummel et al. use graph-based data-flow models. These two approaches can be combined to speed up clone retrieval. Funaro et al. detect Type-III clones. They also use a textual approach on the source code to remove uninteresting code. Agrawal et al. can detect clones for code written in C only. Semantic techniques have limitations as follows. 1) PDG-based techniques are not scalable to large systems. 2) PDG-based techniques need a PDG generator. 3) Graph matching is expensive in PDG-based techniques.

Chapter Summary

Software clones occur due to several reasons such as code reuse by copying pre-existing fragments, coding style and repeated computation using duplicated functions with slight changes in the used variables, data structures or control structures. If we edit a code fragment, we have to check for all related code clones to see if they need to be modified as well. Removal, avoidance and refactoring of cloned code are other important issues in software maintenance. In this chapter, we have discussed software clone detection techniques and tools in depth. We also discuss, compare and analyze the state-of-the-art tools, introduce a taxonomy, and discuss the tools that have not been discussed in previous literature reviews. We select some common clone detectors that were covered by previous surveys. We also add clone detectors that are not covered by previous work. Several tools
<table>
<thead>
<tr>
<th>Tool</th>
<th>Recall</th>
<th>Precision</th>
<th>Portability</th>
<th>Scalability</th>
<th>Robustness</th>
<th>Comparison Method</th>
<th>Language Independence</th>
<th>Language supported</th>
<th>Speed</th>
<th>RAM</th>
<th>Types of clones</th>
<th>Domain</th>
</tr>
</thead>
<tbody>
<tr>
<td>CloneDr</td>
<td>Low</td>
<td>High</td>
<td>Low</td>
<td>Low</td>
<td>Low</td>
<td>Tree/Tree matching algorithm</td>
<td>Needs a parser</td>
<td>C, C++, Java, COBOL</td>
<td>Medium</td>
<td>Low</td>
<td>Type-I, II</td>
<td>Clone detection</td>
</tr>
<tr>
<td>LD</td>
<td>Low</td>
<td>Low</td>
<td>Low</td>
<td>Low</td>
<td>High</td>
<td>Tree/Suffix-tree based algorithm then using Index-based clone detection algorithm</td>
<td>Language independent</td>
<td>Java, C++, Java</td>
<td>N/A</td>
<td>N/A</td>
<td>Type-II</td>
<td>Clone Detection</td>
</tr>
<tr>
<td>CCFinder</td>
<td>High</td>
<td>Low</td>
<td>Medium</td>
<td>High</td>
<td>Low</td>
<td>Token/Suffix-tree based token matching</td>
<td>Needs lexer and trans. rules</td>
<td>C, C++, Java, COBOL</td>
<td>Medium</td>
<td>Medium</td>
<td>Type-I, II and few Type-III</td>
<td>Clone and plagiarism detection</td>
</tr>
<tr>
<td>Dup</td>
<td>High</td>
<td>Low</td>
<td>Medium</td>
<td>High</td>
<td>Low</td>
<td>Token/Suffix-tree based token</td>
<td>Needs lexer</td>
<td>C, C++, Java</td>
<td>Medium</td>
<td>Medium</td>
<td>Type-I, II</td>
<td>Clone and plagiarism detection</td>
</tr>
<tr>
<td>Duploc</td>
<td>Low</td>
<td>Low</td>
<td>High</td>
<td>Low</td>
<td>High</td>
<td>Depends on comparison algorithms</td>
<td>Needs parser</td>
<td>Independent</td>
<td>N/A</td>
<td>N/A</td>
<td>Type-I, and few Type-II</td>
<td>Clone detection</td>
</tr>
<tr>
<td>NICAD</td>
<td>High</td>
<td>High</td>
<td>Low</td>
<td>Low</td>
<td>Low</td>
<td>Depends on comparison algorithms</td>
<td>Needs parser</td>
<td>Java</td>
<td>Medium</td>
<td>Medium</td>
<td>Type-I,III</td>
<td>Clone detection</td>
</tr>
<tr>
<td>DECKARD</td>
<td>Low</td>
<td>High</td>
<td>Low</td>
<td>High</td>
<td>Low</td>
<td>Depends, how comparison is done</td>
<td>Hybrid/Tree-matching</td>
<td>C, Java</td>
<td>Medium</td>
<td>Low</td>
<td>Type-I,II,III</td>
<td>Clone Detection</td>
</tr>
<tr>
<td>CLAN</td>
<td>Medium</td>
<td>Medium</td>
<td>Low</td>
<td>High</td>
<td>High</td>
<td>Metrics/21 function metrics</td>
<td>Need a tool (Dutrix)</td>
<td>C</td>
<td>High</td>
<td>High</td>
<td>Type-I,Weak Type-II,III</td>
<td>Clone and plagiarism detection</td>
</tr>
<tr>
<td>iClones</td>
<td>Low</td>
<td>Medium</td>
<td>Low</td>
<td>Medium</td>
<td>High</td>
<td>Tokens/Smith-Waterman algorithm</td>
<td>Needs parser</td>
<td>Java</td>
<td>Medium</td>
<td>Low</td>
<td>Type-I,II, Weak Type-III</td>
<td>Clone Detection</td>
</tr>
<tr>
<td>CDSW</td>
<td>High</td>
<td>Low</td>
<td>Low</td>
<td>Medium</td>
<td>Medium</td>
<td>Tokens/Smith-Waterman algorithm</td>
<td>Needs lexer</td>
<td>Java</td>
<td>Medium</td>
<td>Low</td>
<td>Type-I,II,III</td>
<td>Clone detection</td>
</tr>
<tr>
<td>Coarse-grained SorcererCC</td>
<td>Low</td>
<td>Low</td>
<td>Low</td>
<td>High</td>
<td>High</td>
<td>Block/Group blocks based on hash values</td>
<td>Needs parser</td>
<td>Java</td>
<td>High</td>
<td>High</td>
<td>Type-I,II</td>
<td>Clone detection</td>
</tr>
<tr>
<td>Our Contribution Approach1</td>
<td>Medium</td>
<td>High</td>
<td>Low</td>
<td>Medium</td>
<td>High</td>
<td>Hybrid/Hash block and similarity measure</td>
<td>Needs parser</td>
<td>Java</td>
<td>High</td>
<td>High</td>
<td>Type-I,II,III</td>
<td>Clone and plagiarism detection</td>
</tr>
<tr>
<td>Our Contribution Approach2</td>
<td>High</td>
<td>High</td>
<td>Low</td>
<td>High</td>
<td>High</td>
<td>Blocks/Classification algorithms</td>
<td>Needs parser</td>
<td>Java</td>
<td>Medium</td>
<td>Medium</td>
<td>Type-I,II,III,IV</td>
<td>Clone, obfuscation and plagiarism detection</td>
</tr>
<tr>
<td>Our Contribution Approach3</td>
<td>High</td>
<td>High</td>
<td>Low</td>
<td>High</td>
<td>High</td>
<td>Blocks/Ensemble classifiers</td>
<td>Needs parser and compiler</td>
<td>Java</td>
<td>High</td>
<td>High</td>
<td>Type-I,II,III,IV</td>
<td>Clone, obfuscation and plagiarism detection</td>
</tr>
</tbody>
</table>
have been excluded since they are not widely used. We compare and classify techniques and tools considering the types of clones they can detect, the granularity of code fragments they analyze, the transformation they perform on code fragments before comparison, the manner in which they represent code fragments to perform comparison, the method used for comparison, the complexity of the comparison process, the outputs they produce, and general advantages and disadvantages.
Acknowledgment

This chapter is based on the paper “A Survey of Software Clone Detection Techniques”, written in collaboration with Jugal Kalita, that proceeding of International Journal of Computer Applications, 0975-8887.
CHAPTER III

CODE CLONE DETECTION USING A COARSE AND FINE-GRAINED HYBRID APPROACH

Introduction

A coarse-grained approach detects clones of methods, statement blocks or classes. In contrast, a fine-grained approach detects clones of sequences of tokens, lines or statements. Our objective is to combine these two approaches. We perform a two-stage analysis which involves coarse detection, followed by fine-grained detection. We use coarse-grained detection to get an overview of clones in terms of blocks and fine-grained detection for detailed analysis. A coarse-grained analysis is used to detect Type-I and Type-II clones and the fine-grained analysis is used to detect Type-III clones.

A coarse-grained technique has high precision since it detects fewer candidate clones than a fine-grained technique. A fine-grained technique has high recall since it detects more reference clones than a coarse-grained technique. The reason we use a fine-grained technique as the second stage is because the first stage, the coarse-grained approach, detects
only a few clones. We combine the two techniques to improve both recall and precision for a dataset. Existing text-based and token-based detection approaches produce many false positives. On the other hand, existing AST-based and PDG-based approaches require much time for transforming the source code into ASTs and PDGs and compare them [76].

We implement the proposed method and evaluate it by using Murakami’s benchmark dataset [76]. Murakami’s references represent code clones with information including where gaps of code clones start and where they end. In contrast, Bellon’s benchmark dataset [11] does not have information about where gaps are.

The rest of the chapter is organized as follows. In The Proposed Method Section, the proposed method is discussed in detail. The similarity measures are described in Similarity Measures Section. Experiment design is discussed in Experiment Design Section. We discuss the experiments we perform in Experiments Section. Discussions on our approach are covered in Discussions Section. Finally, the chapter is concluded in Chapter Summary Section.

**The Proposed Method**

We hash normalize blocks and compare them to detect Type-I and Type-II clones. We use two similarity measures to detect Type-III clones. Details are given later in this section. The proposed method consists of the following steps.

**Step 1.** Lexical analysis and normalization.

**Step 2.** Detecting blocks and extracting sub-blocks.

**Step 3.** Grouping and hashing normalized blocks.
Figure 3.1: The proposed code clone detection method. Each step is illustrated as we analyze the code in two files for the existence of clones.

### Step 1: Detect blocks and extract sub-blocks

<table>
<thead>
<tr>
<th>File 1</th>
<th>File 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>886</td>
<td>665</td>
</tr>
<tr>
<td>private static class $ extends TextAction {</td>
<td>static class NotifyAction extends TextAction {</td>
</tr>
<tr>
<td>887</td>
<td>666</td>
</tr>
<tr>
<td>public $();</td>
<td>NotifyAction();</td>
</tr>
<tr>
<td>888</td>
<td>667</td>
</tr>
<tr>
<td>super($());</td>
<td>super($());</td>
</tr>
<tr>
<td>889</td>
<td>668</td>
</tr>
<tr>
<td>public void actionPerformed(ActionEvent e) {</td>
<td>public void actionPerformed(ActionEvent e) {</td>
</tr>
<tr>
<td>890</td>
<td>669</td>
</tr>
<tr>
<td>JTextComponent target = getFocusedComponent();</td>
<td>JTextComponent target = getFocusedComponent();</td>
</tr>
<tr>
<td>891</td>
<td>670</td>
</tr>
<tr>
<td>if (target instanceof JFormattedTextField) {</td>
<td>if (target instanceof JTextField) {</td>
</tr>
<tr>
<td>892</td>
<td>671</td>
</tr>
<tr>
<td>if (target instanceof JFormattedTextField) {</td>
<td>JTextField field = (JTextField) target;</td>
</tr>
<tr>
<td>893</td>
<td>672</td>
</tr>
<tr>
<td>ftf.setValue(ftf.getValue());</td>
<td>field.postActionEvent();</td>
</tr>
<tr>
<td>894</td>
<td>673</td>
</tr>
<tr>
<td>public boolean isEnabled() {</td>
<td>public boolean isEnabled() {</td>
</tr>
<tr>
<td>895</td>
<td>674</td>
</tr>
<tr>
<td>JTextComponent target = getFocusedComponent();</td>
<td>JTextComponent target = getFocusedComponent();</td>
</tr>
<tr>
<td>896</td>
<td>675</td>
</tr>
<tr>
<td>return (target instanceof JFormattedTextField);</td>
<td>if (target instanceof JTextField) {</td>
</tr>
<tr>
<td></td>
<td>676</td>
</tr>
<tr>
<td></td>
<td>return false;</td>
</tr>
<tr>
<td></td>
<td>677</td>
</tr>
</tbody>
</table>

### Step 2: Perform lexical analysis

#### File 1

- Class/Method blocks by
  - Step 2: Identify similar Class/Method blocks by using Levenshtein Distance or
    - Cost formula:
      - \( \text{cost}_{XY} = \max(\text{Len}(B_X), \text{Len}(B_Y)) \times (1 - \text{LevDist}(B_X, B_Y)) \)

#### File 2

- Class/Method blocks by
  - Step 2: Identify similar Class/Method blocks by using Levenshtein Distance or
    - Cost formula:
      - \( \text{cost}_{XY} = \max(\text{Len}(B_X), \text{Len}(B_Y)) \times (1 - \text{LevDist}(B_X, B_Y)) \)

### Step 3: Group & hash blocks

<table>
<thead>
<tr>
<th>Hash</th>
<th>Block</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td>20</td>
<td>2</td>
</tr>
</tbody>
</table>

### Step 4: Map Similar Sequences to the original source code

<table>
<thead>
<tr>
<th>Hash</th>
<th>Block</th>
</tr>
</thead>
<tbody>
<tr>
<td>30</td>
<td>3.6</td>
</tr>
<tr>
<td>40</td>
<td>4.7</td>
</tr>
<tr>
<td>50</td>
<td>5</td>
</tr>
<tr>
<td>60</td>
<td>8</td>
</tr>
</tbody>
</table>

### Step 5: Identify gapped lines between two Class/Method blocks

<table>
<thead>
<tr>
<th>Hash</th>
<th>Block</th>
</tr>
</thead>
<tbody>
<tr>
<td>70</td>
<td>9</td>
</tr>
<tr>
<td>80</td>
<td>10</td>
</tr>
<tr>
<td>90</td>
<td>11</td>
</tr>
</tbody>
</table>

### Step 6: Map Similar Sequences to the original source code

<table>
<thead>
<tr>
<th>Hash</th>
<th>Block</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td>20</td>
<td>2</td>
</tr>
</tbody>
</table>

### References

- Blocks: File 1
  - Block1: Line: 886-896
    - Method: class $ extends TextAction
  - Block2: Line: 665-677
    - Method: static class NotifyAction extends TextAction

- Blocks: File 2
  - Block3: Line: 887-888, 891-893
    - Method: $();
  - Block4: Line: 666-677
    - Method: super($());

- Blocks: File 1
  - Block5: Line: 686-697
    - Method: $();
  - Block6: Line: 666-667
    - Method: super($());

- Blocks: File 2
  - Block7: Line: 668-672
    - Method: super($());
  - Block8: Line: 673-676
    - Method: super($());

- Blocks: File 1
  - Block9: Line: 889-890
    - Method: super($());
  - Block10: Line: 670-672
    - Method: super($());

- Blocks: File 2
  - Block11: Line: 675-677
    - Method: super($());

- Tokens: File 1
  - Tokens: File 2
Step 4. Detecting similar blocks using Levenshtein Distance/Cosine similarity.

Step 5. Identifying gapped lines.

Step 6. Mapping similar blocks to the original source code.

Algorithm 1 Comparison of Two Blocks using Levenshtein Distance

1: \textbf{procedure} \textit{LevenshteinDistance}(B_1, B_2)
2: \hspace{1em} define \( LD[n+1][m+1] \)
3: \hspace{1em} set \( LD[i][0..m] \leftarrow i \)
4: \hspace{1em} set \( LD[0..n][j] \leftarrow j \)
5: \hspace{1em} \textbf{for} \( j \leftarrow 0, n \ \textbf{do} \)
6: \hspace{2em} \( b_1 \leftarrow B_1\.getLine(i-1) \)
7: \hspace{1em} \textbf{for} \( j \leftarrow 0, m \ \textbf{do} \)
8: \hspace{2em} \( b_2 \leftarrow B_2\.getLine(j-1) \)
9: \hspace{2em} \textbf{if} \( b_1 = b_2 \ \textbf{then} \)
10: \hspace{3em} \( score \leftarrow 0 \)
11: \hspace{2em} \textbf{else} \)
12: \hspace{3em} \( score \leftarrow 1 \)
13: \hspace{2em} \textbf{end if} \)
14: \hspace{1em} \( LD[i][j] \leftarrow \text{Min}(LD[i-1][j]+1, \)
15: \hspace{1em} \( LD[i][j]+1, \)
16: \hspace{1em} \( LD[i-1][j-1] + \text{cost} \)
17: \hspace{1em} \textbf{end for} \)
18: \textbf{return} \( Sim = 1 - \frac{LD(b_1,b_2)}{\text{Max}(\text{len}(b_1,b_2))} \times 100 \)
19: \textbf{end procedure} \)

Algorithm 2 Type-III Clone Detection

1: \textbf{procedure} \textit{CloneDetection}(BlockFragments)
2: \hspace{1em} \textit{Clones} \leftarrow 0
3: \hspace{1em} \textbf{for} \( i \leftarrow 0, \text{BlockFragments}\_\text{Length} \ \textbf{do} \)
4: \hspace{2em} \textbf{for} \( j \leftarrow 0, \text{BlockFragments}\_\text{Length} \ \textbf{do} \)
5: \hspace{3em} \textit{Cosine}(B_i, B_j)
6: \hspace{2em} \textbf{if} \( \text{Sim} \geq \tau \% \ \textbf{then} \) \( \text{//} \tau = 70\% \)
7: \hspace{3em} \textit{Clones} \leftarrow \textit{Clones} + 1
8: \hspace{2em} \textbf{end if} \)
9: \hspace{1em} \textbf{end for} \)
10: \textbf{end for} \)
11: \textbf{return} \textit{Clones} \)
12: \textbf{end procedure} \)
To explain our steps, we use the two program fragments given in Figure 3.1(a) as running example.

**Lexical Analysis and Normalization**

The first step is to transform and normalize all source files into special token sequences to detect not only identical clones but also similar ones. This also helps in dealing with varying numbers of whitespaces occurring together. Figure 3.1(a) gives the original files and 3.1(b) gives the two program fragments after lexical analysis and normalization. Identifiers have been replaced by the $ sign.

**Detecting Blocks and Extracting Sub-Blocks**

This step needs not only lexical analysis but also syntactic analysis to detect every block from the given source files. All blocks, including classes, methods and block statements, are extracted using the Java Development Tool (JDT)\(^1\). Figure 3.1(c) shows the detected blocks for the two files. For example in File 1, it has detected a class block between lines 886 and 896, and a method block from line 887 to line 888.

**Grouping and Hashing Normalized Blocks**

After identifying all normalized blocks, we group them into similar blocks such as class blocks, method blocks, loop statements, branch statements and assignment statements. This helps detect similar clones later using Levenshtein distance or cosine similarity. These two similarity measures are discussed in Similarity Measures Section. In

\(^1\)http://www.eclipse.org/jdt/
Figure 3.1(d), we see the blocks detected in File 1 on top and blocks from File 2 at bottom in each group.

Next, this step calculates a hash value of the text of a block. We use `HashCode()` in Java as the hash function, which is simply a number, a 32-bit signed `int`. This step can find both of Type-I and Type-II clones by looking for two blocks or statements that have the same hash value. This happens if their text representations after normalization are equal to each other. For example, in File 1, a method block between lines 887 to line 888 has been detected as a Type-I or Type-II clone of a method block between lines 666 to line 667 in File 2.

**Detecting Similar Blocks Using Levenshtein Distance/Cosine Similarity**

Normalized blocks, which are similar, are detected using the Levenshtein distance algorithm. Levenshtein distance measures the distance between two blocks, which is the minimal number of insertions, deletions, and substitutions that will transform one block into the another. We also use cosine similarity. These two are discussed in detail in Similarity Measures Section.

**Identifying Gapped Lines**

After similar blocks are detected in Step 4, we use a string differences algorithm to detect gaps between blocks and identify line-level gaps.
Mapping Similar Blocks to the Original Source Code

All of the code and similar blocks that are detected in Steps 3 and 4 are mapped back to the source code, using by file path, start line and end line.

Algorithm 3 Comparison of Two Blocks using CosineSim

1: procedure CosineSimilarityScore(Block1, Block2)
2: for i ← 0, BlockLines1.Length do
3:     if Line.length < 0 then
4:         if FreqVector.containsKey(Line) then
5:             freq1 ← value1 + 1
6:             freq2 ← value2
7:         else
8:             UniqueLines.add(Line);
9:         end if
10:     end if
11: end for
12: for i ← 0, BlockLines2.Length do
13:     if Line.length < 0 then
14:         if FreqVector.containsKey(Line) then
15:             freq1 ← value1
16:             freq2 ← value2 + 1
17:         else
18:             UniqueLines.add(Line);
19:         end if
20: end if
21: end for
22: for i ← 0, UniqueLines.Length do
23:     VectorBlock1 ← VectorBlock1 + freq1 × freq1
24:     VectorBlock2 ← VectorBlock2 + freq2 × freq2
25: end for
26: return Sim(VectorBlock1, VectorBlock2)← \frac{\sqrt{\text{VectorBlock1}} \times 100}{\sqrt{\text{VectorBlock1}} \times \sqrt{\text{VectorBlock2}}}
27: end procedure

Similarity Measures

We use two similarity measures: 1) Levenshtein similarity and 2) Cosine similarity in Step 4 of our approach discussed in The Proposed Method Section. We use these two metrics for detecting Type-III clones.
Levenshtein Similarity

Levenshtein distance is named after the Russian scientist Vladimir Levenshtein, who proposed this algorithm [65]. It is a metric for measuring the difference between two sequences. It is one of the most widely used algorithms to calculate edit distance. We use Levenshtein distance to detect similar clones in two blocks of code. If the Levenshtein similarity (Eq.3.1) is above a threshold value, we declare two fragments, i.e., ($B_1$ and $B_2$) are candidate Type-III clone pairs.

\[
Similarity = 1 - \frac{LevDist(B_i, B_j)}{\max(\text{Len}(B_i), \text{Len}(B_j))} \times 100
\]

(3.1)

where $LevDist$ is the Levenshtein distance and $\text{Len}(B_i)$ and $\text{Len}(B_j)$ are the lengths of two blocks in numbers of lines. The complexity of Algorithm 1 is $O(m \times n)$, where $n$ and $m$ are the lengths of $B_i$ and $B_j$.

Cosine Similarity

We also use another measure of similarity, which is cosine similarity. Cosine similarity between two vectors measures the cosine of the angle between them. The bigger the return value, the more similar the two code fragments. Our approach converts each block of code to a vector. The value of cosine similarity (Eq.2) between two code fragments is compared against a constant threshold value, to decide whether two fragments, $B_1$ and $B_2$, should be reported as candidate clone pairs.

\[
\text{CosSim}(v_1, v_2) = \cos(\alpha) = \frac{\sum_{i=1}^{n} v_{1i} \times v_{2i}}{\sqrt{\sum_{i=1}^{n} v_{1i}^2} \times \sqrt{\sum_{i=1}^{n} v_{2i}^2}}
\]
Table 3.1: Target software systems

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Language</th>
<th>#Lines</th>
<th>#files</th>
<th>#methods</th>
<th>#References/Clones</th>
</tr>
</thead>
<tbody>
<tr>
<td>Netbeans</td>
<td>Java</td>
<td>14,360</td>
<td>97</td>
<td>972</td>
<td>55</td>
</tr>
<tr>
<td>Eclipse-ant</td>
<td>Java</td>
<td>34,744</td>
<td>161</td>
<td>1754</td>
<td>30</td>
</tr>
<tr>
<td>Javax-swing</td>
<td>Java</td>
<td>204,037</td>
<td>414</td>
<td>10,971</td>
<td>777</td>
</tr>
<tr>
<td>Eclipse-jdtcore</td>
<td>Java</td>
<td>147,634</td>
<td>741</td>
<td>7,383</td>
<td>1,345</td>
</tr>
</tbody>
</table>

(3.2)

where $\text{CosSim}$ is the cosine similarity between two vectors $\mathbf{v}_1$ and $\mathbf{v}_2$ and $\alpha$ is the angle between them.

**Experiment Design**

To compare our approach with other detectors in detecting Type-I and Type-II clones, we choose eight (CloneDr [10], LD [45], CCFinder [49], Dup [6], Duploc [28], Deckard [46], CDWS [76], and Coarse-grained [44]) detectors and depend on results reported by Hotta et al. [44]. To compare our approach with other detectors in detecting Type-III clones, we also choose eight (CloneDr [10], CLAN [71], CCFinder [49], Dup [6], Duploc [28], Nicad [91], Deckard [46], and CDSW [76]) detectors and use results reported by Murakami et al. [76]. To evaluate our tool, we use source code of four Java projects. Details of the source codes used are given in Table 2.1. Our implementation handles programs in Java only because we use the JDT tool for development. We perform two experiments to answer the following research questions.

**RQ1:** Is the proposed method more accurate than existing detectors for Type-I and Type-II clones?
**RQ2:** Does the proposed method have higher *precision* and *F-measure* than existing detectors for Type-III clones?

**RQ3:** Does the proposed method have higher *precision* and *F-measure* than existing detectors for all of Type-I, Type-II, and Type-III clones?

We use the following terms in evaluating our results. A *reference* is a clone pair that is included in the reference or the true clone set. A *candidate* is a clone pair that is detected by clone detectors. The *contained* metric we use is also used in the study of Bellon *et al.* [11]:

\[
\text{contained}(CF_1, CF_2) = \frac{|\text{lines}(CF_1) \cap \text{lines}(CF_2)|}{|\text{lines}(CF_1)|}
\]  

(3.3)

where is $CF_1$ and $CF_2$ refer to the set of lines of code in code fragment $CF$.

We also use the metric *ok* value to indicate whether a candidate subsumes a reference.

\[
\text{ok}(CP_1, CP_2) = \min(\max(\text{contained}(CP_1, CF_1, CP_2, CF_1), \text{contained}(CP_2, CF_1, CP_1, CF_1)), \max(\text{contained}(CP_1, CF_2, CP_2, CF_2), \text{contained}(CP_2, CF_2, CP_1, CF_1))))
\]

(3.4)

where is $CP.CF_1$ and $CP.CF_2$ are two code fragments when a candidate clone subsumes a reference clone and satisfies the following condition:

\[
\text{ok}(CP_1, CP_2) \geq \text{threshold}.
\]
The *good* value metric is defined by Bellon et al. [11], to indicate whether a candidate sufficiently matches a reference. The *good* value metric is stronger than the *ok* value metric [11]. However, we only use the *ok* value metric because it is enough to detect Type-I,II and III clones. We say $CP_1$, $CP_2$ are clones of each other if the *ok* metric is satisfied. We use 0.7 as the threshold, which is also used in Bellon et al.'s study [11], to evaluate the accuracy of detected clones for a given target software system and a given detector $D$. $Cands$ refers to a set of clone pairs, $Refs$ refers to the set of the clone references and $DetectedRefs$ refers to the set of the clone candidates. The following formulas define precision, recall, and $F$-measure.

$$\text{Precision}(S,D) = \frac{|DetectedRef(S,D)|}{|Cands(S,D)|}$$  \hspace{1cm} (3.5)

$$\text{Recall}(S,D) = \frac{|DetectedRef(S,D)|}{|Refs(S)|}$$  \hspace{1cm} (3.6)

$$F\text{-measure} = \frac{2 \times \text{Recall} \times \text{Precision}}{\text{Recall} + \text{Precision}}$$  \hspace{1cm} (3.7)

**Experiments**

We perform two experiments on four target systems that are shown in Table 3.1. The purpose of the first experiment is to determine whether Steps 2 and 3 of our approach produce good recall and higher precision than existing clone detectors. The second experiment is to discover whether Steps 4 and 5 of our approach produce higher precision
and F-measure than existing clone detectors or not. We use Murakami’s dataset\(^2\), which consists of clone references with gaps [75].

\(^2\)http://sdl.ist.osaka-u.ac.jp/h-murakm/2014_clone_references_with_gaps/
Figure 3.3: The results for Type-III.

(a) **Recall**

(b) **Precision**

(c) **F-Measure**
Figure 3.4: The results for Type-I, II and III.

(a) Recall

(b) Precision

(c) F-Measure


**Experiment A**

Table 3.1 shows the number of detected clone pairs from the Type-I and Type-II clone references [44]. In this experiment, we choose detectors that were used in the experiment of Hotta et al. [44] for comparison with our approach. We calculate recall, precision and F-measure for our approach. For Step 1 of our approach, which is lexical analysis and normalization, we use open source code available at github\(^3\).

Figure 3.2a shows the comparison of recall of all the clone detectors for Type-I and Type-II clone references. CCFinder is the best among all the clone detectors for Eclipse-ant and Javax-swing datasets. LD is the best for Eclipse-jdtcore and Netbeans datasets based on Hotta’s results. Our approach cannot achieve highest recall but is not the lowest in all cases.

Our approach achieves highest precision compared with others. Figure 3.2b shows the values of precision of all the clone detectors for Type-I and Type-II clone references. Our approach gets first position for Netbeans, Eclipse-jdtcore, and Javax-swing datasets. It gets the third position for Eclipse-ant dataset because the Eclipse-ant dataset has only a few reference clones and some of these clones are divided into two parts: one part in one block and the second part in another block.

Figure 3.2c shows the values of F-measure for all the clone detectors. Our approach gets the first position for Eclipse-jdtcore and Javax-swing datasets. It gets second position in Eclipse-ant and Netbeans datasets. It achieves a good balance of recall and precision for the Type-I and Type-II clones references.

\(^3\)https://github.com/k-hotta/ECTEC
Experiment B

The goal of Experiment B is to answer questions RQ2 and RQ3. In this experiment, we choose the detectors used in the Murakami et al.’s [76] experiment to compare with our approach using Levenshtein distance and cosine similarity. We also calculate recall, precision and F-measure of our approach.

Figure 3.3a shows the comparison of recall for all the clone detectors for Type-III clone references. CCFinder is the best among the clone detectors. The median and average of CCFinder are the best. The median and average of our approach are in the middle behind CCFinder, CDSW, Dup, and Deckard. Figure 3.3b shows the case of precision. Our approach using Levenshtein distance ranks first in precision and our approach using cosine similarity gets the second position. We conclude that our approach achieves high precision compared with other detectors. Figure 3.3c shows the comparison of F-measure. The median and average of our approach using Levenshtein distance gets first position and our approach using cosine similarity gets second position. The value of precision of our approach using Levenshtein distance or cosine similarity is high and the value F-measure of our approach either using Levenshtein distance or cosine similarity is high. Figure 3.3b and 3.3c show that our approach in both cases is the best in precision and F-measure. Therefore, we achieve our objective and answer RQ2 positively.

Figure 3.4a shows the recall for all the clone detectors for Type-I, II and III clones. The median of CCFinder is still the best among all the clone detectors. The median of our approach cannot achieve the highest recall but we conclude that our approach does not have the lowest recall. Figure 3.4b shows the value of precision. Our approach using
Figure 3.5: Cosine similarity speed.

Levenshtein distance ranks second in precision after CDSW, and our approach using cosine similarity gets the third position out of eight detectors in this case. CloneDr is the fourth position. We conclude that our approach achieves high precision for Type-I, II, and III clones. Figure 3.4c shows F-measure. Both median and average of our approach in both cases gets the first and second positions. Figure 3.4c also shows our approach in both cases is the best in F-measure. Therefore, we answer RQ3 positively as well.

We measure the execution time of our approach using Cosine similarity. Figure 3.5 shows the execution time obtained by varying thresholds to detect clones in the dataset. Our approach using cosine similarity can detect clones in a few seconds to about 10 minutes. Also, we measure our approach using Levenshtein distance. Figure 3.6 shows the execution time obtained by varying thresholds to detect clones in the dataset. Our approach using Levenshtein distance can detect clones in several seconds to about 15 minutes. Therefore, our approach can detect clones in a short time.
Figure 3.6: Levenshtein distance speed.

Table 3.2: Comparison the medians and the averages of our approaches for Type-III clones results with other existing approaches. The existing detectors results are obtained from Murakami et al. [76]. The best entries are in boldface.

<table>
<thead>
<tr>
<th>Tool</th>
<th>Recall (%)</th>
<th>Precision (%)</th>
<th>F-measure (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>High</td>
<td>Median</td>
<td>Average</td>
</tr>
<tr>
<td>CloneDr</td>
<td>10.80</td>
<td>8.55</td>
<td>17.55</td>
</tr>
<tr>
<td>CLAN</td>
<td>34.50</td>
<td>28.05</td>
<td>29.65</td>
</tr>
<tr>
<td>CCFinder</td>
<td>100.00</td>
<td>79.40</td>
<td>73.40</td>
</tr>
<tr>
<td>Dup</td>
<td>100.00</td>
<td>61.50</td>
<td>68.25</td>
</tr>
<tr>
<td>Duploc</td>
<td>100.00</td>
<td>29.40</td>
<td>39.70</td>
</tr>
<tr>
<td>DECKARD</td>
<td>88.90</td>
<td>53.85</td>
<td>56.03</td>
</tr>
<tr>
<td>NICAD</td>
<td>100.00</td>
<td>53.85</td>
<td>56.03</td>
</tr>
<tr>
<td>CDSW</td>
<td>100.00</td>
<td>68.85</td>
<td>72.25</td>
</tr>
<tr>
<td>Our Approach using Levenshtein Distance</td>
<td>64.25</td>
<td>50.76</td>
<td>53.94</td>
</tr>
<tr>
<td>Our Approach using Cosine Similarity</td>
<td>68.75</td>
<td>56.83</td>
<td>56.22</td>
</tr>
</tbody>
</table>

Table 3.3: Comparison the medians and the average of our approaches for all Type-I, II and III clones results with other existing approaches. The existing detectors results are obtained from Murakami et al. [76]. The best entries are in boldface.

<table>
<thead>
<tr>
<th>Tool</th>
<th>Recall (%)</th>
<th>Precision (%)</th>
<th>F-measure (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>High</td>
<td>Median</td>
<td>Average</td>
</tr>
<tr>
<td>CloneDr</td>
<td>48.10</td>
<td>21.69</td>
<td>71.20</td>
</tr>
<tr>
<td>CLAN</td>
<td>56.10</td>
<td>37.31</td>
<td>38.43</td>
</tr>
<tr>
<td>CCFinder</td>
<td>100.00</td>
<td>83.95</td>
<td>85.03</td>
</tr>
<tr>
<td>Dup</td>
<td>80.20</td>
<td>73.20</td>
<td>71.20</td>
</tr>
<tr>
<td>Duploc</td>
<td>46.70</td>
<td>25.00</td>
<td>24.18</td>
</tr>
<tr>
<td>DECKARD</td>
<td>85.40</td>
<td>69.00</td>
<td>65.85</td>
</tr>
<tr>
<td>NICAD</td>
<td>76.30</td>
<td>49.20</td>
<td>47.35</td>
</tr>
<tr>
<td>CDSW</td>
<td>79.00</td>
<td>51.93</td>
<td>55.20</td>
</tr>
<tr>
<td>Our Approach using Levenshtein Distance</td>
<td>62.73</td>
<td>45.66</td>
<td>47.68</td>
</tr>
<tr>
<td>Our Approach using Cosine Similarity</td>
<td>62.58</td>
<td>48.16</td>
<td>48.89</td>
</tr>
</tbody>
</table>
Discussions

Clone References

It is hard to check for true clones by manual comparison in a target software system. Therefore, for fair comparison, we use datasets of Murakami and Bellon, which recent detectors have used, to compare our approach with others. Murakami et al.’s clone references [75], reannotate the clone references of Bellon et al. [11] with information about gapped lines. A change in clone references can affect the results of precision.

Hashing Collision

We use hash values, which as mentioned earlier, are computed using the `hashcode()` method that produces a product sum over the entire text of the string, for comparing two blocks of code. We use the same hash function, which is supported by a Java library, that Hotta et al. [44] use in their approach. The difference between their approach and ours is that our approach classifies blocks as class blocks, method blocks, if statement blocks, and for statement blocks and hashes these normalized blocks separately for reducing the chances of collision.

Different Programming Languages in Target Software Systems

Because we use the Java Development Tool (JDT) that parses only Java code, we are able to perform experiments in this study for Java projects only. It is possible to perform more experiments in C and other programming languages also to judge how our approach extends to them.
Figure 3.7: How detection of Type-III clones changes as we change Levenshtein distance threshold between $35\% \leq \tau < 100\%$.

**Thresholds for Levenshtein Distance and Cosine Similarity**

In this study, we choose the threshold of similarity between two blocks to be $35\% - 99\%$. When we apply $100\%$ threshold value, Type-I and Type-II clones are detected. With less than $35\%$ threshold value, some *Type-III* clones are missed or more false positives clones are detected. We apply different threshold values for Levenshtein distance and Cosine similarity computations as shown in Figures 3.7 and 3.8, respectively. We conclude that the best range for the threshold $\tau \geq 60\%$ for Levenshtein distance and $\tau \geq 80\%$ for cosine similarity. We compare the medians and averages of our results with the other existing tools for Type-III clones as shown in Table 3.2 and Figure 3.3. We also compare the median and average of our results with the other existing tools for Type-I, II and III clones as shown in Table 3.3 and Figure 3.4.
Figure 3.8: How detection of Type-III clones changes as we change Cosine similarity threshold between $35\% \leq \tau < 100\%$.

Chapter Summary

This section has presented a hybrid clone detection technique that first uses a coarse-grained technique to improve precision and then a fine-grained technique to get more information about clones and to improve recall. We use hash values and grouping of blocks to detect Type-I and Type-II clones, and Levenshtein distance and cosine measures for blocks to detect gapped code clones (Type-III). Our experimental results indicate that our method achieves high precision and F-measure in most cases compared to other detectors. In this chapter, we demonstrate the following.

- Normalizing blocks followed by grouping and hashing helps detect Type-I and Type-II clones.
- We use two similarity measures to detect Type-III clones and tailor the Levenshtein distance algorithm to use for code clone detection. Levenshtien distance is a string metric for measuring the distance between two sequences. The tailored Levenshtien
distance algorithm can measure distance between lines of code. We also use cosine similarity, tailored to measure angular distance between lines, represented as vectors.

- We demonstrate that our proposed method has higher precision and F-measure than existing methods.
Acknowledgment

This chapter is based on the paper “Code clone detection using coarse and fine-grained hybrid approaches”, written in collaboration with Jugal Kalita, that in proceeding of IEEE International Conference on Intelligent Computing and Information Systems (ICICIS), 2015 IEEE Seventh International Conference on (pp. 472-480). IEEE.
CHAPTER IV

MACHINE LEARNING FRAMEWORK FOR DETECTING SEMANTIC CODE CLONES

Introduction

In the software engineering life cycle, maintenance is the most expensive and time-consuming phase. In a large software system, pairs of code segments often occur in different locations, but are functionally identical or similar. Sloppy or even good programmers find it easy to make minor modifications to an existing code segment to serve the current purpose in some other part of a program or a project. Very often programmers find sets of useful statements, called code blocks, and copy-paste them as necessary, modifying as per requirement to make the software development process faster. Duplicated code blocks are popularly known as code clones (CC). Research has reported that 7%-23% of large software projects are code clones [8], [90]. Many studies show that a software system with frequent occurrence of code clones is difficult to maintain [94]. One of the problems with code cloning is when an original code block, which is cloned, contains a bug, causing
ripple effects in all cloned blocks, distributed all over the program or project. Detecting code clones is an important and challenging task. Automatic detection of clones not only improves the software maintenance task, but also may be useful in identifying software plagiarism [114] and code obfuscation [97], detection of malicious software [31], discovery of context-based inconsistencies [47] and opportunities for code refactoring [4].

Automatic clone detection is an active area of research. A number of efforts to detect clones effectively have been published. Existing clone detection methods commonly use similarity metrics to compare fragments of code. All published methods have difficulty in detecting semantic clones, the most challenging types of clones. Semantic clones are syntactically different, but functionally they produce similar outcomes. Traditional approaches are ineffective because their similarity metrics do not capture semantics well. As a result, performance of the methods become fairly low in terms of various assessment metrics. Machine learning has been recently used successfully in several approaches for automatic detection of code clones, although the amount of work is limited. Moreover, attempts at using machine learning so far have not done well in addressing the issue of semantic clones.

We present a novel machine learning framework for automated detection of all four types of clones using features extracted from ASTs and PDGs, and using code block pairs as examples. We introduce a formal way to model the code clone detection problem, and use state-of-the-art classification models to assess the prediction performance of our scheme. Experimental results demonstrate that our approach outperforms existing clone detection methods in terms of prediction accuracy.

We organize this chapter as follows. Types of clones are discussed in Types of Clones Section. In Machine Learning for Pairwise Clone Detection Section, we propose a new ma-
chine learning framework for detection of semantic code clones. We evaluate and compare our proposed method and report in Experimental Evaluation section. Finally, we conclude our work in Chapter Summary Section.

**Types of Clones**

Code clone detection may be performed within a single program or project, or across programs or projects. A modular program usually consists of a set of sub-programs or methods. A method is a set of executable program statements with precisely defined starting and ending points, performing a cohesive task. In this chapter, we term it a method block. A method block may be divided into sub-blocks (e.g. loops, conditional statements, etc.). In our work, we use the terms method block and code block interchangeably.

**Definition 1 (Block).** A block \( B \) is a sequence of statements, \( S_i, i = 1, \cdots, M \), comprising of programming language specific executable statements such as loops, logical statements and arithmetic expressions:

\[
B = \langle S_1, \cdots S_M \rangle.
\]

**Definition 2 (Corpus or Dataset).** A corpus \( C \) of blocks or a dataset is a set of \( N \) blocks extracted from a single program or project, or a collection of programs or projects:

\[
C = \{ B_1, \cdots, B_N \}.
\]
Definition 3 (Code Clones). Two code blocks \( B_i \) and \( B_j \) from a corpus \( C \) constitute a code clone pair if they are similar based on some metric:

\[
\text{clone}(B_i, B_j) = \begin{cases} 
1, & \text{if } \text{sim}(B_i, B_j) > \theta \\
0, & \text{otherwise.}
\end{cases}
\] (4.1)

We measure similarity considering a set of characteristics or features we use to describe a block. We can describe a block simply in terms of the statements contained in it, or in terms of other characteristics extracted from the statements in the code, as we will see later. \( B_i \) and \( B_j \) are clones, if they score high using a pre-specified similarity criterion (\( \text{sim} \)).

The code clone detection problem can be defined as follows.

Definition 4 (Code Clone Detection). Given a pair of blocks \( B_i \) and \( B_j \in C \) where \( C \) is a corpus of blocks, code clone detection is a boolean mapping function \( f : B_i \times B_j \rightarrow N \in [1, 0] \), where \( B_i \times B_j \) represents the similarity function given in Equation 4.1.

To detect if a pair of blocks are clones of each other, two kinds of similarities may be considered. Blocks \( B_i \) and \( B_j \) may be textually similar or may functionally perform similar tasks or the same task, without being textually similar. The first kind of clones is simple in nature, usually resulting from the practice of copying and direct pasting. However, the second type of similarity is difficult to define precisely. Bellon et al. [11] identified three types of clones based on textual similarity of the programs.

Definition 5 (Type-I: Exact Clones). Two blocks are the exact clones of each other if they are exactly the same except whitespaces, blanks and comments.
Let $B_i$ and $B_j$ be two blocks from a corpus $C$. Let $B_i = <S_{i1}, \ldots, S_{iN_i}>$, and $B_j = <S_{j1}, \ldots, S_{jN_j}>$. Let $B_i^t = \text{trim}(B_i)$ where $\text{trim}(.)$ is a function that removes whitespaces, blanks and comments from the block and its statements. Thus, whitespaces that cover an entire line are removed, as well as whitespaces within statements. $B_i$ and $B_j$ are exact clones of each other if i) $|B_i^t| = |B_j^t|$, i.e., they are both of the same length after trimming, and ii) $\forall k, k = 1, \ldots, |B_i^t| S_{ik}^t \equiv S_{jk}^t$ where $\equiv$ means that the two statements are exactly the same, considered as strings. The superscript $t$ means after trimming.

**Definition 6** (Type-II: Renamed Clones). Two blocks are the renamed clones of each other if the blocks are similar except for names of variables, identifiers, types, literals, layouts, whitespaces, blanks and comments.

Let $B_i^n$ and $B_j^n$ be two trimmed and normalized blocks: $B_i^n = \text{norm}(\text{trim}(B_i))$ and $B_j^n = \text{norm}(\text{trim}(B_j))$ where $\text{norm}(.)$ is a literal normalization function. Normalization replaces all the variables from $B_i$ and $B_j$ with a single generic variable name, among other operations.

Formally, $B_i$ and $B_j \in C$ where $C$ is a corpus are renamed clones if i) $|B_i^n| = |B_j^n|$, i.e., they are both of the same length after trimming and normalizing, and ii) $\forall k, k = 1, \ldots, |B_i^n| S_{ik}^n \equiv S_{jk}^n$.

**Definition 7** (Type-III: Gapped clones). Two copied blocks are gapped clones if they are similar, but with modifications such as added or removed statements, and the use of different identifiers, literals, types, whitespaces, layouts and comments.

The new flexibility introduced is the addition or removal of statements. Assume we are given two blocks $B_i$ and $B_j$ from a corpus $C$, and let $B_i^n$ and $B_j^n$ be their trimmed and normalized versions, as described earlier. Two gapped sequences can be aligned using
various techniques that generate an alignment score \( (\text{ascore}) \) for each alignment \([29, 80]\). The value of \( \text{ascore} \) is obtained by considering the costs of gaps, and the costs of character mismatch and replacement between the two strings.

We say \( B_i \) and \( B_j \) are gapped clones of each other if \( \text{ascore}(B_i^n, B_j^n) > \theta \) for a user-defined threshold \( \theta \). We can make things a little simpler by considering the blocks (original, trimmed or normalized) as bags or sets instead of sequences, as is quite frequently the case in natural language processing \([70]\). In such a situation, we can approximate a version of gapped similarity \( (\text{ascore'}) \) between blocks \( B_i^n \) and \( B_j^n \) as follows:

\[
\text{ascore'}(B_i^n, B_j^n) = \frac{|B_i^n \cap B_j^n|}{|B_i^n \cup B_j^n|},
\]

giving the fraction of statements that are common between the two blocks, both being considered as sets. Again, we can say \( B_i \) and \( B_j \) are gapped clones of each other if \( \text{ascore'}(B_i^n, B_j^n) > \theta \) for a user-defined threshold \( \theta \). By definition, \( 1 \geq \text{ascore'}(\cdot, \cdot) \geq 0 \).

The fourth type of clones is semantic clones. Semantic clones are the most challenging types of clones. Instead of comparing program texts which is relatively easy to do, semantic clones are difficult to identify as they deal with the meaning or purpose of the blocks, without regards to textual similarity. A real life example of semantic clones is a pair of obfuscated blocks or programs \([60]\), where syntax-wise the blocks are by and large different from each other, but the overall meanings of both are the same.

**Definition 8** (Type-IV: Semantic clones). Two blocks are semantic clones, if they are semantically similar without being syntactically similar. In other words, two blocks \( B_i \) and
$B_j$ are semantic clones if

$$\text{semsim}(B_i, B_j) = \text{semsim}(B_i^n, B_j^n) > \theta,$$

(4.3)

where $\text{semsim}(., .)$ is a semantic similarity function.

The idea of semantic similarity is not easy to grasp because it requires some level of understanding of the meanings of programs, whether formal or otherwise. The formal semantics of a program or a block can be described in several ways, the predominant ones being denotational semantics, axiomatic semantics and operational semantics [37, 109]. Denotational semantics composes the meaning of a program or a block by composing it from the meaning (or denotation, a mathematical expression or function) of its components in a bottom-up fashion. Axiomatic semantics defines the meaning of a program or block by first defining the meanings of individual commands by describing their effects on assertions about variables that represent program states, and then writing logical statements with them. Operational or concrete semantics does not attach mathematical meanings to components within a program or block, but describes how the individual steps of a block or program takes place in a computer-based system on some abstract machine. No matter which approach is used for describing formal semantics, the meaning of a block or program is obtained from the meanings ascribed to the individual components. To obtain the semantics of a block or a program, it is initially parsed into syntactic or structural components, and for each syntactic component, its corresponding meaning is obtained, and finally the meaning of the block is put together from these components, following appropriate rules.
Thus, we could say two blocks $B_i$ and $B_j$ are semantic clones if

$$\text{semsim}(B_i^n, B_j^n) = \text{semsim}([B_i^n], [B_j^n]) > \theta,$$  \hspace{1cm} (4.4)

where $[B]$ gives the meaning of a block $B$, possibly following one of the methods discussed earlier. In practice, we should note that the semantics of a block may be computed without resorting to formal semantics.

Different types of clones are illustrated with the help of a few simple programs in Figure 1.1. The original code block, in the center of the figure, swaps values of two integer variables using a temporary variable. The Type-I clone is an exact replica of the original code block or program. In case of a Type-II clone, only a few of the literals are changed. The gapped clone block is a replica of the original except that a line has been deleted. The Type-IV clone block (top right) shows another approach to swap two different variables without using a third variable. Structurally, the code blocks are dissimilar; however because the purpose of both code blocks is the same, semantically they are similar. On the other hand, the Type-I through III clone blocks are structurally similar although what they do are different.

**Machine Learning for Pairwise Clone Detection**

A straightforward approach to determine if two code blocks are semantically similar without necessarily being syntactically similar may proceed as follows: Trim and normalize the two blocks as discussed earlier, obtain the formal semantics of the two blocks using a method alluded to earlier; and, compare the formal semantic representations using Equation
6.2. However, tools to obtain formal semantics are not readily available. In addition, formal semantic representations are strings themselves, requiring additional string comparisons. It is also unclear that formal semantic representations will add substantially to efficient and effective code clone detection. Thus, it may be appropriate to investigate if other approaches may work well in detecting if two blocks of code are semantic clones of each other.

Code clone detection has been treated as a pairwise similarity analysis problem, where two blocks are clones if a given block is similar to the given reference block. However, machine learning usually considers individual samples for training and predicts class labels. Instead of comparing the structural and meaning representations (which may be long and/or difficult-to-obtain strings themselves) directly, to compare if two blocks are syntactically or semantically similar, we can extract relevant characteristics of the blocks by looking at selected portions of them or other associated structures like ASTs and PDGs; these are usually called features in the machine learning literature. To apply machine learning to pairwise clone detection, we use features of both the reference and target blocks.

**Definition 9** (Pairwise Learning). Given a set of $N$ pairs of training samples, each sample (a pair of blocks) labelled with a clone type depending on their mutual similarity, a classification model can act as a mapping function $f : X \to Y$, where $X$ is an unknown pair of code blocks and $Y$ is the possible clone type predicted by the model. Training samples are represented as feature vectors, $\text{features}(<B_i, B_j>) = <f_1, f_2, \ldots, f_M, C_k>$ of size $M$, created by combining the features of two different blocks $(B_i, B_j)$ and a clone type, $C_k$ associated with $(B_i, B_j)$, forming a training sample matrix of size $N \times (M + 1)$. 
When a block is represented as a set of features, the semantics of a block $B^n_i$ is described as given below:

$$[B^n_i] \approx <f_{i1}, \cdots f_{ik}>$$

(4.5)

where $\approx$ means an approximation. Thus, a block’s semantics can be simply represented as a list of features; of course this is not a precise representation of semantic meaning.

Equation 4.3 can now be restated as:

$$semsim(B^n_i, B^n_j) =$$

$$semsim(<f_{i1}, \cdots f_{ik}>, <f_{j1}, \cdots f_{jk}>) > \theta. \quad (4.6)$$

That is, similarity between two blocks is measured by computing similarity between the two feature based representations.

Thus, instead of using one of the approaches to describing the formal semantics of a program block, we use features of PDGs for semantic representation. We use other features obtained from ASTs as well. In our work, we additionally combine a few so-called traditional features, as discussed later.

**AST and PDG: Novel Features for Clone Detection**

We pre-process the blocks by trimming and normalizing as discussed earlier. We extract basic characteristics, which we term Traditional Features, like Lines of Code (LOC), number of keywords, variables, assignments, conditional statements and iteration state-
ments [52] used in a given piece of source code. Traditional features alone are inadequate in capturing the syntactic and semantic characteristics of a block.

Syntactic similarity between two blocks of code is also likely to impact upon the similarity in meanings of the blocks, and hence we also parse the blocks into their structural components in terms of Abstract Syntax Tree (AST). Each node of the tree represents a construct occurring in the given source code. Leaf nodes of the tree contain variables used in the code. Unlike majority of published clone detection methods that compare the two syntactic trees directly, we compute certain characteristics or features extracted from the ASTs, which we call syntactic features. Figure 4.1 shows an example AST created by the AST Generator software we use. We traverse the AST in post-order manner and extract only non-leaf nodes containing programming constructs such as Variable Declaration Statements (VDS), While Statements (WS), Cast Expressions, Class Instances, and Method Invocations. Next, we represent frequencies of these programming constructs as AST features in a vector.

The PDG features can be called semantic or meaning features. PDGs make explicit both the data and control dependence for each operation in a program. Data dependencies represent the relevant data flow relationships of a program. Control dependencies represent the essential control flow relationships. A sample PDG derived from a code block is illustrated in Figure 4.2. Edges represent the order of execution of program nodes. We parse the AST, created by an AST generator (GenerateAST) further to create an implicit PDG structure and extract features. In other words, we do not construct an explicit PDG but extract the features we could have extracted from an explicit PDG. We use the same post-order traversal of AST and find the frequencies of various dependency relationships...
between different constructs. We consider total 12 constructs and compute 43 relationships among them up to level three, and use them as our PDG or semantic features. For example, the feature $Expr \cdot Assign \cdot Decl$, captures the number of dependency relationships that occur sequentially as Expression, followed by Assignment and then followed by Declaration statements in the given code.

In Algorithm 4, we describe the feature extraction scheme. $L_{AST}$ and $L_{PDG}$ are the lists of pre-specified AST attributes and PDG attributes (please refer to Appendix A for details) and count their frequencies in the post-order sequence of the tokens (non-leaves) extracted by PostOrderTokens and stored in $V$. We avoid leaf token nodes as leaf nodes of ASTs contain only variables. Frequencies of AST and PDG attributes are stored as features in a vector $F$. In case the AST feature’s MatchToken matches a pre-specified AST attribute, we increase the count of that attribute or feature. The method DependencyFreq checks for the occurrence of the PDG attribute $L_{PDG}$, in vector $V$, and returns the frequency of such relationship in $V$. Please refer to Appendix A for the details about the features extracted during the process.

The features of PDG we extract include dependence information among parts of the code. We extract data dependency features that count the occurrence of declaration, expression and assignment, which are defined as hierarchical ordering observed in the PDG. We also extract control dependency features that count the occurrence of the data dependency features. Examples of such features are the number of Assignments that come after Declarations, obtained by counting the occurrence of the assignments which are dependent on declarations; the number of Declarations coming after Control (e.g. $i < count$, for, while, if, switch, etc.), obtained by counting the occurrence of the declarations which are dependent
Figure 4.1: Example of AST derived from code block. MD: MethodDeclaration VDS: VariableDeclarationStatement, WS: WhileStatement, CE: ConditionalExpression, E: Expression

Figure 4.2: Program dependency graph showing control and data dependency among the statements.
Algorithm 4 AST & PDG Feature Extraction

1: INPUT : $B$ // Target method block
2: OUTPUT : $F = \{f_{AST_1}, \cdots, f_{AST_N}, f_{PDG_1}, \cdots, f_{PDG_M}\}$ // Set of $N$ AST and $M$ PDG features
3: Steps :
4: $T \leftarrow \phi$ // AST root node
5: $L_{AST} = \{A_1 \cdots A_N\}$ // List of $N$ AST attributes
6: $L_{PDG} = \{P_1 \cdots P_M\}$ // List of $M$ PDG attributes
7: $V \leftarrow \text{PostOrderTokens}(T)$ //Store post order sequence of non-leaf nodes in vector $V$
8: //Counting frequency of AST features
9: for $i = 1 \cdots |L_{AST}|$ do
10: for $j = 1 \cdots |V|$ do
11: if $\text{MatchToken}(A_i, V_j)$ then
12: $f_{AST_i} = f_{AST_i} + 1$
13: end if
14: end for
15: $F = F \cup f_{AST_i}$
16: end for
17: //Counting frequency of PDG features
18: for $i = 1 \cdots |L_{PDG}|$ do
19: $f_{PDG_i} \leftarrow \text{DependencyFreq}(P_i, V)$
20: $F = F \cup f_{PDG_i}$
21: end for
22: return $F$

on control statements; the number of times a nested iteration occurs; the number of times a nested selection occurs, and so on

We combine features of ASTs and PDGs for finding syntactic and semantic clones effectively since alone they may not be sufficient. Considering the three types of features we have discussed in this section, we now represent a block in terms of three types of features. Although it is not strictly semantics any more, we say the "semantics" of a trimmed and normalized code block $B^n_i$ is described as given below:

$$[B^n_i] \approx <f^t_{i_1}, \cdots, f^t_{i_k}, f^s_{i_1}, \cdots, f^s_{i_k}, f^m_{i_1}, \cdots, f^m_{i_k}>.$$ (4.7)
In this equation, we denote the three sets of features with different superscripts: \( t \) for traditional features, \( s \) for syntactic features, and \( m \) for semantic or meaning features, (which are actually PDG based features), and separate the three groups with vertical lines, for clear separation. In our work, we generated a total 100 of features, combining the three different types. The distribution of feature categories is shown in Figure 4.3.

**Fusion of Block Features**

We combine feature vectors (Equation 4.7) extracted from a pair of target and reference code blocks to create the training dataset. We fuse the sequence of features from the two different blocks. Although there are three types of features in the description of a block, to simplify the notation, we can rewrite Equation 4.7, without distinguishing among the feature types, as

\[
[B^n_i] \approx features(B_i) = \langle f_{i1}, \cdots f_{ik} \rangle
\]  

(4.8)

where \( k = k_t + k_s + k_m \). Similarly,

\[
[B^n_j] \approx features(B_j) = \langle f_{j1}, \cdots f_{jk} \rangle.
\]  

(4.9)
Given two blocks $B_i$ and $B_j$, and their clone label $C_l$, the combined feature vector, 

$features(<B_i, B_j>)$ can now be represented as a fused feature vector. We fuse the two vectors in three different ways as discussed below.

**Linear Combination:** We concatenate the two feature vectors. Simple concatenation gives rise to a fused feature vector of size $2k + 1$. Linear combination looks like as follows:

$$features(<B_i, B_j>) = <f_{i1}, \ldots, f_{ik}, f_{j1}, \ldots, f_{jk}, C_l>, \quad (4.10)$$

where $C_l$ is the class label (type of clone) for the pair. A linear combination results in double the number of features. To reduce the size, we may use two other simple combination approaches.

**Multiplicative Combination:** Here we combine two different feature sequences by multiplying the corresponding feature values:

$$features(<B_i, B_j>) = <f_{i1} \cdot f_{j1}, \ldots, f_{ik} \cdot f_{jk}, C_l>. \quad (4.11)$$

**Distance Combination:** Nearness, the opposite of distance, is the most obvious way to calculate the similarity between two block features. We use the absolute difference between two feature values to fuse the features of a pair of blocks:

$$features(<B_i, B_j>) = <|f_{i1} - f_{j1}|, \ldots, |f_{ik} - f_{jk}|, C_l>. \quad (4.12)$$
Clone Detection Framework

Our scheme is similar to a traditional machine learning framework. We have two phases, training and testing. In training, we use labelled pairs of cloned blocks from a given hand-curated code clone corpus. All method blocks are detected from the given corpus using lexical and syntactic analysis. We extract method blocks and perform preprocessing steps, including trimming and normalization. Next, we generate ASTs and PDGs of the blocks and extract features from them. Following Equation 4.7, we create a complete feature vector for each block by combining traditional, AST and PDG features. We fuse feature vectors of two target blocks by using one of the Equations 4.10, 4.11 or 4.12. All the above steps are iterated for all possible pairs of blocks for creating a training dataset for the classification model. For identifying the possible clone type of unlabeled code blocks, we perform the same sequence of steps to create a fused feature vector of the two given blocks and pass it through the classifier for prediction of the possible clone type. Figure 4.4 demonstrates the work-flow of our approach.

Experimental Evaluation

In this section, we evaluate and establish the superiority of our proposed machine learning framework for detecting all types of clones. In our experiments, we use only methods extracted from Java source code as a corpus for training and testing. However, this model is general in nature and can be extended easily to any other high level programming language. Our primary goal is to improve clone detection accuracy for all types of clones with a special emphasis on semantic clone detection. We use a large number of existing
Figure 4.4: Workflow of the proposed clone detection framework.

classification algorithms and compare the effectiveness of the proposed framework with the state-of-the-art detection methods based on their reported results.

Datasets

We use IJaDataset 2.0 [100], a large inter-project Java repository containing source code from 25,000 open-source projects, with 3 million source files, 250 million lines of code, from SourceForge and Google Code. This benchmark was built by mining IJaDataset for functions. The published version of the benchmark considers 44 target functionalities [50].

For this experiment, we consider all types of clone lengths in IJaDataset 2.0 that are 6 lines or 50 tokens or longer, which is the standard minimum clone size for benchmarking [11, 95]. There is no agreement on when a clone is no longer syntactically similar, and the authors claim that it is also hard to separate the Type-III and Type-IV clones in the
IJaDataset [100]. As result, some prior researchers have divided Type-III and Type-IV clones into four classes based on their syntactic similarity [95] as follows: Very Strongly Type-III (VST3) clones are ones that have a syntactic similarity in the range [90% - 100%), Strongly Type-III (ST3) in [70% - 90%), Moderately Type-III in [50% - 70%) and Weakly Type-III/Type-IV (WT3/4) in (0%-50%], where means ( exclusive and ] means inclusive range.

Classification Models

We train and test our proposed framework using fifteen state-of-the-art classification models, starting from the popularly used Naïve Bayes [48] model to a recently published gradient boosting tree model, Xgboost (eXtreme Gradient Boosting) [16]. While selecting the various classification models, we try to keep a balance among different learning approaches, including probabilistic and non-probabilistic, generative and discriminative, linear and non-linear, regression, decision trees and distance based models.

Naïve Bayes [48] is a simple probabilistic classifier based on applying Bayes’ rule. Linear Discriminant Analysis (LDA) [73] is commonly used as a dimensionality reduction technique in pre-processing for pattern-classification and machine learning applications. Support Vector Machine (SVM) [30] is a maximum margin classification model. LogitBoost [33] is a boosting classification algorithm. LogitBoost and AdaBoost are close to each other in that both perform additive logistic regression. In addition, we use several tree ensemble models including Extra Trees [36], Rotation Forest [89] coupled with Principal Components Analysis (PCA), Random Forest [13] and Random Committee [110]. Instance Based Learner (IBK) [3] is similar to a k-Nearest Neighbor algorithm. Bagging [12]
is an ensemble meta-estimator that fits base classifiers each on random subsets of the original dataset and then aggregates their individual predictions. We also use decision tree algorithms such as J48 [96] and Random tree [62] for our experimentations. Random Subspace [42] selects random subsets of the available features to be used in training the individual classifiers in an ensemble. Xgboost [16] is a recently proposed fast and accurate boosting tree model that uses a loss function when constructing individual trees. The classification models used in our work are summerised in Table 4.1.

We represent a pair instance as one vector as explained above. Similar blocks are detected using one of the classification algorithms. We compare the outcomes of all the classifiers discussed above to avoid bias towards any particular classifiers, as our main emphasis is on effective feature generation, not the classification model. Classifiers are trained and tested using cross-validation with 10 folds. We ensure balance between match and non-match classes in each fold to the same as proportion found in the overall dataset.

<table>
<thead>
<tr>
<th>Classifiers</th>
<th>Model Characteristics</th>
<th>Platform</th>
</tr>
</thead>
<tbody>
<tr>
<td>Naïve Bayes</td>
<td>Probabilistic, makes independence assumption among features</td>
<td>Weka 3.8</td>
</tr>
<tr>
<td>LDA</td>
<td>Finds a linear combination of features as separator between classes</td>
<td>Weka 3.8</td>
</tr>
<tr>
<td>LIBLINEAR/SVM</td>
<td>Linear maximum margin classifiers</td>
<td>Weka 3.8</td>
</tr>
<tr>
<td>Sequential Minimal Optimization (SMO)</td>
<td>Quadratic programming solver for SVMs</td>
<td>Weka 3.8</td>
</tr>
<tr>
<td>IBL</td>
<td>K nearest-neighbor classifier, can choose k using cross-validation</td>
<td>Weka 3.8</td>
</tr>
<tr>
<td>J48</td>
<td>An implementation of C4.5 decision tree classifier</td>
<td>Weka 3.8</td>
</tr>
<tr>
<td>Random Tree</td>
<td>A decision tree classifier that uses k random attributes at each node</td>
<td>Weka 3.8</td>
</tr>
<tr>
<td>Extra Tree</td>
<td>A decision tree classifier, works with numeric attributes</td>
<td>Weka 3.8</td>
</tr>
<tr>
<td>Boostap Aggregation (Bagging)</td>
<td>Ensembles classifier that creates a classifier from separates samples of the training dataset</td>
<td>Weka 3.8</td>
</tr>
<tr>
<td>LogitBoost</td>
<td>A statistical implementation of Adaboost, a meta-learning algorithm</td>
<td>Weka 3.8</td>
</tr>
<tr>
<td>Random Subspace</td>
<td>Ensemble classifier that creates multiple decision trees constructed using randomly chosen features</td>
<td>Weka 3.8</td>
</tr>
<tr>
<td>Random Committee</td>
<td>An ensemble of randomize base classifier</td>
<td>Weka 3.8</td>
</tr>
<tr>
<td>Rotation Forest</td>
<td>An ensemble of classifiers created by making k subset of features, running PCA on each subset, and keeping all principal components</td>
<td>Weka 3.8</td>
</tr>
<tr>
<td>Random Forest</td>
<td>Ensemble Decision Tree</td>
<td>Weka 3.8</td>
</tr>
<tr>
<td>Xgboost</td>
<td>Ensemble classifier that creates boosted decision trees, each with an associated objective function that is optimized</td>
<td>R 3.3.0</td>
</tr>
</tbody>
</table>
Evaluation

We generate extensive results to assess the robustness of our proposed model in detecting semantic clones along with all other type of clones. We experiment with a varying number of features and with different data instances to show that the our features are able to achieve a high detection accuracy. We report only the best performing classifiers for most of the experiments and compare them with state-of-the-art clone detection methods. However, for more results, one can refer to the Appendix A. To generate AST from a given block in order to extract features we use Eclipse Java Development Tools (JDT).

Performance of Different Classifiers

We randomly select 20K pair instances from IJaDataset. To compare the three feature fusion methods and the performance of classifiers, we run our all the classifiers three times. Figure 4.5 shows the comparison of all fifteen classifiers using linear, multiplicative, and distance combinations respectively. Experimental results show that the tree ensemble methods such as Rotation Forest, Random Forest and Xgboost achieve better outcomes among all the classifiers. This is because tree ensemble approaches create many trees with samples and random attributes correct the errors in the parent trees to generate next level of trees. For example, Xgboost has higher performance because it has a regularization component to reduce overfitting.
Varying data size and feature types

We assess the importance of combining Traditional, AST and PDG features in various ways and report the results from IJaDataset in Figure 4.6. We create four subsets of IJaDataset using 5K, 10K, 15K and 20K instances from each class. Results produced by three best performing classifiers reported above with varying data sizes, show that the performance of the classifiers improves substantially as we combine both syntactic and semantic features to detect clones. Interestingly, the performance of classifiers using semantic features is consistent irrespective of data sizes and fusion methods. We also observe that distance and multiplicative combinations produce better results than linear combination for all sizes of data.

Experimenting with varying feature sizes

We perform two different kinds of experiments with varying numbers of features, selecting equal numbers of features from each feature type (Traditional, AST and PDG) and using feature selection methods (Figure 4.8). The intention behind such experiments is to show the significance of our proposed features in achieving better accuracy, and that
Figure 4.6: Performance of three best classifiers with syntactic and semantic features.

(a) Distance combination

(b) Multiplicative combination

(c) Linear combination
Figure 4.7: Learning curve: Performance of Random Forest and Xgboost with varying features.

It is not by chance. The growing learning curve (Figure 4.7) clearly indicates that the detection accuracy improves with the increase in the numbers of features. We also notice that Xgboost using multiplicative combination achieves higher performance than others. We use two feature selection algorithms namely Gain Ratio [83] and Information Gain [82]. For each experiment, we use different sizes of the feature sets ranked by the feature selection algorithms. Similar to the learning curve based on randomly selected feature sets, judiciously selected feature sets also show a growing trend in the performance. This further establishes the fact that our features are crucial in deriving high accuracy in detection results.

To avoid implementation bias, we compare the performance of our method with contemporary clone detection methods, using their reported results on IJaDataset. Different methods have reported a range of Precision, Recall and F-score values. We show the
Figure 4.8: Performance of Random Forest and Rotation Forest with varying number of features using Gain Ratio and InfoGain feature selection algorithms.

(a) Distance combination

(b) Multiplicative combination

(c) Linear combination
maximum value of the range for reported results putting, the classifiers in the best highs possible. Interestingly, a majority of the detection methods are incapable of detecting semantic clones or Type-IV clones. Figure 4.9 shows comparison of our results with the state-of-the-art detectors based on recall and F-measure. From the results, it is evident that NiCad performs better than all other methods in detecting Type-I/II, VST3, and ST3 clones based on the F-measure metric. We report only results for the best performing classifiers for most of the experiments, which are Xgboost and Random Forest with various fusion
types. Results clearly show that our method is effective in detecting Type-IV clones along with other clone types in comparison to the other methods.

**Chapter Summary**

Semantic code clone detection is a challenging task and needs automatization. We propose a machine learning framework for automatic detection of large numbers of code clones. We use, for the first time, a combination of traditional, AST and PDG features instead of performing tree a graph comparisons directly. In order to avoid a-priori bias towards any classification models, we use 15 state-of-the-art classification models to obtain their relative performance using our features. Experimental results clearly indicate that our proposed features are highly valuable to achieve high detection accuracy.

We extend our work to achieve further improvements in the rest of dissertation. For example, by using features of Java ByteCodes obtained by compiling Java programs.
Acknowledgment

This chapter is based on the paper “Semantic Clone Detection Using Machine Learning”, written in collaboration with Jugal Kalita, that in proceeding of IEEE International Conference on Machine Learning and Applications (ICMLA), 2016 15th IEEE International Conference on (pp. 1024-1028). IEEE.
CHAPTER V

AUTOMATED LABELING TYPE-IV CODE CLONE USING JAVA BYTECODE

Introduction

Software maintenance is a critical activity in terms of cost and effort. Many studies show that a software system with many code clones is more difficult to maintain compared to a software system with a fewer clones [86]. Code clone detection taking into account Java ByteCode and other intermediate code representations, and using machine learning has been not been studied earlier. In this chapter, we introduce a new dataset of clone references for Type-IV clones. Bellon et al. [11] created a clone detector, which they call all an oracle; this detector does not deal with Type-IV (semantic) clones. The objective of this chapter is to generate a set of method-level code clones that can help evaluate future code clone detection approaches. In order to enable Type-IV detection, we propose a method to construct clone using Java ByteCcode. To collect better data and also help manual inspection of clone pairs, we build a tool to automatically extract method pairs of interest, along with
their clone types. The benefit of using Java ByteCode is that method pair blocks that may not be syntactically similar at source code level but are in fact semantically similar, can be identified better for labeling as Type-IV (semantic) clones.

The rest of the chapter is organized as follows. Prior research is discussed in Prior Research Section. Characteristics of Java ByteCode are described in Java ByteCode Overview Section. In The Proposed Method Section, the proposed method is discussed in detail. Finally, the chapter is concluded in Chapter Summary Section.

## Prior Research

Bellon et al.’s [11] datasets are widely used as references by others. Their datasets, introduced in 2007, have become the standard references for evaluation of every new clone detector. Bellon’s dataset identifies only Type-I, II and III clones. The approach has three steps. 1) Eight target software systems, namely netbeans, eclipse-ant, eclipse-jdtcor, J2sdk1.4.0-javax-swing, weltab, cook, snns, and Postgresql are selected. 2) Potential code clones are identified by the state-of-the-art clone detectors. Then, the information containing locations of the detected clones were given to experts for actual verification. 3) The experts were sent only 2% of the identified code clones randomly. Each of these potential code clones is checked manually. Bellon’s dataset does not contain gapped lines, and location information for Type-III clones.

Lavoie and Merlo [63] introduced clone detectors for Type-III clones using the Levenshtein metric, without relying on statistical properties. The clones are of size up to a few
MLOCs. The resulting clone references are for two Java software systems, namely tomcat and eclipse. Their work only detects Type-III clones.

Kurtz and Le [59] obtained clones from three open source systems, namely Apache, Python and PostgreSQL. They generated a set of method level semantic code clones with high confidence to assist evaluation of clone detectors. They build a tool to automatically load function pairs to help with manual inspection. The problem with this set of clone references is that its size is small with only 66 method clone pairs of Type-I and 9 pairs of Type-IV.

Murakami et al. [75] extended Bellon’s dataset of clone references by adding information containing locations of gapped lines. Our approach extends Bellon’s and Murakami’s datasets by adding Type-IV clones so that Type-IV clone detectors can be trained and/or tested.

Svajlenko et al. [100] present a big data clone detection benchmark called IJaDataset 2.0, which is a large Java repository containing 25,000 open-source projects from SourceForge and Google Code.

**Java ByteCode Overview**

Java ByteCode is a sequence of instructions for the virtual machine to simulate basic functionalities such as conditions and loops [101]. Each ByteCode contains one or more opcode. ByteCode is between Java source code and actual machine code. Java Virtual Machine takes the ByteCode and converts it into machine code.
Algorithm 5 Automated Labeling Types of Clones

1: \textbf{procedure} \textit{OracleClones}(\textit{MethodCodes})
2: \hspace{1em}\text{\textit{TI/IIClones} $\leftarrow 0$, \textit{TIIIClones} $\leftarrow 0$,}
3: \hspace{1em}\text{\textit{TIVClones} $\leftarrow 0$}
4: \hspace{1em}\textbf{for} \textit{i} $\leftarrow 0$, \textit{MethodCodes.Length} \textbf{do}
5: \hspace{2em}\textbf{for} \textit{j} $\leftarrow \textit{i}+1$, \textit{MethodCodes.Length} \textbf{do}
6: \hspace{3em}\textit{Code}_1 \leftarrow \text{CompileToByteCode}(	extit{MethodB}_i)
7: \hspace{3em}\textit{Code}_2 \leftarrow \text{CompileToByteCode}(	extit{MethodB}_j)
8: \hspace{3em}\textit{SimM} $\leftarrow \text{LevDist}(	extit{MethodB}_i, \textit{MethodB}_j)$
9: \hspace{3em}\textit{SimBCode} $\leftarrow \text{LevDist}($\textit{Code}_1, \textit{Code}_2$)$
10: \hspace{3em}//\mu = 50\%, \pi = 80\%, \alpha = 100\$
11: \hspace{3em}\textbf{if} \textit{SimM} = \alpha \textbf{then}
12: \hspace{4em}\textbf{if} \textit{SimBCode} $\geq \pi$ \textbf{then}
13: \hspace{5em}\textit{TI/IIClones} $\leftarrow \textit{TI/IIClones} + 1$
14: \hspace{4em}\textbf{end if}
15: \hspace{3em}\textbf{else if} \mu \leq \textit{SimM} && \textit{SimM} > \alpha \textbf{then}
16: \hspace{4em}\textbf{if} \textit{SimBCode} $= \pi$ \textbf{then}
17: \hspace{5em}\textit{TypeIIIClones} $\leftarrow \textit{TIIIClones} + 1$
18: \hspace{4em}\textbf{end if}
19: \hspace{3em}\textbf{else} \textit{SimM} < \mu
20: \hspace{4em}\textbf{if} \textit{SimBCode} $= \pi$ \textbf{then}
21: \hspace{5em}\textit{TypeIVClones} $\leftarrow \textit{TIVClones} + 1$
22: \hspace{4em}\textbf{end if}
23: \hspace{3em}\textbf{end if}
24: \hspace{2em}\textbf{end for}
25: \hspace{1em}\textbf{end for}
26: \hspace{1em}\textbf{return} \textit{TI/IIClones}, \textit{TIIIClones}, \textit{TIVClones}
27: \textbf{end procedure}

The Proposed Method

We construct a clone dataset automatically using Java ByteCode, which is platform
independent, representing the unified structure of the code. Such clone pairs are not always
detectable at source code-level, and we have found that Levenshtein similarity metric helps
identify Type-IV clones automatically when ByteCode is used. The proposed method con-
sists of the following steps. Figure 5.1 illustrates the workflow of our approach. To explain
our steps, we use the two method codes given in Figure 5.1(a) as running example.
1. **Analyze lexically, normalize and detect method blocks.** The first step is to transform and normalize all source files into special token sequences to identify not only identical clones but also similar ones. Figure 5.1(b) gives the two method blocks after lexical analysis and normalization. Identifiers have been replaced by $ sign. All method blocks are extracted using the Java Development Tool (JDT). For example in File 1, it has detected a method between lines 2 and 9.

2. **Identify similar method blocks using Levenshtein distance.** This helps identify similar method codes using Levenshtein distance after identifying all normalized codes. We
choose a threshold based on their syntactic similarity as follows: Type-I and II clones have syntactic similarity of 100%, Type-III clones have syntactic similarity [50% - 100%) and Type-IV clones have similarity in (0%- 50%). These percentages are similar to those used by prior researchers [95]. We compare each method and its Java ByteCode against all other methods in the same dataset for clones.

3. **Convert method blocks to Java ByteCode classes.** This step is to convert all method codes into Java ByteCode classes to identify Type-IV (semantic) clones using the Javac compiler. Figure 5.1(c) gives the two Java ByteCode classes after compiling the two methods codes. This can help identify candidate semantic clones, which may be hard to identify.

4. **Filter and remove noise in Java ByteCode classes.** The Java ByteCode classes contain a significant amount of noise. The noise is filtered and removed, including all labels and instruction numbers as shown in Figure 5.1(d).

5. **Identify similar Java ByteCode classes using Levenshtein metric.** This helps identify similar Java ByteCode classes using the Levenshtein metric after filtering and removing all noise. It generates Type-III and Type-IV clones, which may not have been identified at source code level. We choose 50% as a threshold based on their syntactical similarity.

6. **Inspect Manually.** This step is a manual inspection on all pairs of methods detected using software to determine that they are actual clones and their types. We inspect clones independently. We label them with the help of a group of expert Java programmers with Masters and PhD degrees in Computer Science. The details of the datasets are given in Table 5.1. A majority of existing clone datasets used in prior papers are incomplete in nature. They avoid labeling semantic code clones. A brief summery of the
Table 5.1: Brief description of our Java code clone corpus

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Paired Codes</th>
<th>Type-I and II</th>
<th>Type-III</th>
<th>Type-IV</th>
<th>False Agreement (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Staple</td>
<td>152</td>
<td>30</td>
<td>39</td>
<td>25</td>
<td>38</td>
</tr>
<tr>
<td>.netbeans-javadoc</td>
<td>452</td>
<td>54</td>
<td>146</td>
<td>39</td>
<td>213</td>
</tr>
<tr>
<td>eclipse-ant</td>
<td>787</td>
<td>118</td>
<td>392</td>
<td>66</td>
<td>211</td>
</tr>
<tr>
<td>EIRC</td>
<td>870</td>
<td>32</td>
<td>394</td>
<td>146</td>
<td>298</td>
</tr>
<tr>
<td>Sample_j2sdk1.4.0-javax-swing</td>
<td>800</td>
<td>200</td>
<td>282</td>
<td>118</td>
<td>200</td>
</tr>
<tr>
<td>Sample_eclipse-jdtcore</td>
<td>800</td>
<td>200</td>
<td>200</td>
<td>169</td>
<td>231</td>
</tr>
</tbody>
</table>

extended datasets is given in Table 5.1. In the table, the second column indicates how many paired-blocks we extract to expand the existing dataset. Agreement refers to the probability of reliability between observers or raters. We compute Kappa statistic [102] agreement between every pair of observers’ decisions using Equation 5.1 and take the average probability of agreement between all the raters and report the same in the table:

\[ \kappa = \frac{p_o - p_e}{1 - p_e}, \]  

(5.1)

where, \( p_o \) is the relative observed agreement among raters and \( p_e \) is the hypothetical probability of chance agreement.

7. **Store detected Type-IV clones as references clones.** All of the similar codes that are identified in Steps (b), (c), (d), (e) and (f) are stored in the reference clones dataset.

**Chapter Summary**

The goal of this study is to analyze the source code using ByteCode as an intermediate representation, instead of using other intermediate representations. The reason behind using ByteCode is that it is platform independent and represents the unified structure of the code. Such structure is not always detectable at source code-level, and as a result, our tool is more effective than already existing tools. We exploit the benefit of Java ByteCode, to
detect code blocks which might not be syntactically similar at source level but are in fact semantically similar, for generating Type-IV (semantic) clones. Syntactic dissimilarities among different types of loops and selection blocks in the source code have been transformed to unified format at the ByteCode level. As a result, ByteCode can help detect semantic clones which might be hard to detect.
CHAPTER VI

OBFUSCATED CODE DETECTION—AN APPLICATION OF SEMANTIC CLONE DETECTION

Introduction

Code obfuscation is a technique to alter the original content of the code in order to sow confusion. Malware creators use obfuscation to camouflage existing malicious code and make the task of signature based malware detection tools more challenging. Automated code obfuscation tools make malware development a child’s play even for non-professional attackers. Traditional malware detection tools based on apriori known signatures become ineffective due to the morphing of earlier known malicious code. It makes a well-protected system vulnerable to zero-day-attacks more often. The positive use of code obfuscation is equally important for protecting intellectual property rights of proprietary software. Code obfuscation prevents attackers from malicious reverse engineering of sensitive parts of a software project and helps prevent software piracy [19]. However, we concentrate only on the negative aspect of the problem, i.e., detecting malicious code.
The common way to perform camouflaging is to change the syntactic structure of the code, keeping the semantics and functionality of the original malware the same. Obfuscation techniques transform a code block in two different ways, using metamorphism and polymorphism [18]. Metamorphism obfuscates the entire code by inserting certain dead code, and by performing code substitution and code transposition. On the other hand, polymorphism uses transformations to obfuscate loops in the code. Existing malware detectors treat malware code as a sequences of bytes and extract signature to classify them by matching with known malware signatures. Syntactic or structural signatures are weak and ineffective in detecting camouflaged code and are overlooked easily by signature based malware detectors. Effective anti-malware software based on semantic structure of the code is a current need to mitigate the issue of ever-evolving variants of known malware.

In this work, we focus only Java code and develop a machine learning framework for effective detection of code obfuscation. We model obfuscated code detection as a kind of semantic code clone detection. We use syntactic and semantic features of pairs of original and target codes for detection of possible obfuscated code.

The rest of the chapter is organized as follows. In Background Section, background is discussed in detail. Prior research on code obfuscation is described in Prior Research Section. Our code clone detection framework is discussed in An Integrated Detection Framework Section. We discuss the experiments we perform in Experimental Evaluation Section. Finally, the chapter is concluded in Chapter Summary Section.
Background

Code obfuscation is a form of semantic code cloning where two malicious pieces of code may be structurally or syntactically dissimilar, but semantically behave in a similar way. Below we define code obfuscation in a more formal way. We draw a resemblance between code obfuscation and semantic code clones. Two pieces of code are semantic clones if they are functionally similar. Precisely defining the term semantic similarity between two pieces of code is hard. In comparison to syntactic similarity, which compares program texts and is relatively easy to do, semantic similarity is difficult to identify as it deals with the meaning or purpose of the codes, without regards to textual similarity.

The idea of semantic similarity is not easy to grasp because it requires some level of understanding the meanings of programs, whether formal or otherwise. The formal semantics of a program or a piece of code can be described in several ways, the predominant ones being denotational semantics, axiomatic semantics and operational semantics [37, 109]. Denotational semantics composes the meaning of a program or a fragment of code by composing it from the meaning (or denotation, a mathematical expression or function) of its components in a bottom-up fashion. Two pieces of code have the same meaning if their composed denotations are the same. Axiomatic semantics defines the meaning of a program or code fragment by first defining the meanings of individual commands by describing their effects on assertions about variables that represent program states, and then writing logical statements with them. In this paradigm, two pieces of code that write an algorithm slightly differently but produce the same results are considered semantically equivalent, provided their initial assertions are the same. Operational or concrete semantics does not attach
mathematical meanings to components within a program or code fragment, but describes how the individual steps of a piece of code or program takes place in a computer-based system on some abstract machine. No matter which approach is used for describing formal semantics, the meaning of a code fragment or program is obtained from the meanings ascribed to the individual components. To obtain the semantics of a code fragment or program, it is initially parsed into syntactic or structural components, and for each syntactic component, its corresponding meaning is obtained, and finally the meaning of the piece of code is put together from these components, following appropriate rules. Thus, we could say two pieces of code $C_i$ and $C_j$ are semantically similar if

$$\text{SemSim}(C_i, C_j) = \text{SemSim}^*(\llbracket C_i \rrbracket, \llbracket C_j \rrbracket) > \varphi,$$

where $\text{SemSim}^*(\ldots)$ is a formal measure of similarity between the two pieces of code. $\llbracket C_i \rrbracket$ is the formal semantics of code fragment $C_i$ computed following a formal approach.

In this paper, we will not delve deeper into how $\llbracket C_i \rrbracket$, $\llbracket C_j \rrbracket$ or, $\text{SemSim}^*(\ldots)$ may be computed exactly following a semantic theory. In practice, we approximate the computation of $\text{SemSim}(C_i, C_j)$ using other means as discussed in this chapter. In other words, the primary focus in this chapter is an approximate computation of $\text{SemSim}^*(\ldots)$ using non-formal semantic means. Assuming, we can provide a good computational approximation to $\text{SemSim}^*(\ldots)$, we can proceed to define semantic clones. We call this approximation $\text{SemSim}^*(\ldots)$.

**Definition 10** (Semantic Clone). A piece of code $C_i$ is a semantic clone of another piece of code $C_j$ (or vice versa) if they are semantically similar without being syntactically similar.
Abstract definition of obfuscated code can be obtained using the notion of syntactic similarity in the following way.

**Definition 11** (Code Obfuscation). A piece of code $C_j$ is obfuscated version of another piece of code $C_i$ if they exhibit similar functionality although structurally they are different from each other. They are similar to semantic clones. In other words, obfuscated code pairs can be represented using a discrete function as follows.

\[
\text{Obfuscated}(C_i, C_j) = \begin{cases} 
1, & \text{if } \text{SemSim}(C_i, C_j) > \varphi \\
\text{SynSim}(C_i, C_j) < \varphi; & \\
0, & \text{otherwise},
\end{cases}
\]

where $\text{SemSim}(., .)$ is a semantic similarity function, and $\varphi$ is a user set threshold.

Several effective methods have been proposed to discover obfuscated code. A brief sketch of these methods is given below.

**Prior Research**

During the last few decades a significant amount of research has been performed in both areas. A number of effective tools and computational methods have been developed to address the issues of code clone detection and detecting obfuscated code independently. We discuss prior research on clone detection methods in Chapter II, followed by obfuscated code detection.

Obfuscated code detection methods are abundant in the literature. Likarish *et al.* [67] propose an obfuscation detection scheme in JavaScript scripts. Automatic analysis of
malware behavior using machine learning is proposed by Rieck et al. [87]. Their approach is to detect classes of malware with similar behavior using clustering and then assigning unknown malware to a class using classification.

Wang et al. [106] propose a technique to detect malicious JavaScript in web pages using deep features extracted by Stacked Denoising Auto-encoders (SdA) for classification. Results are promising although SdA is slow in training. O’Kane et al. [78] present a method using an optimal set of operational codes (opcodes) from an assembly language or machine language.

Ragkhitwetsagul et al. [84] study and evaluate 30 tools using two experimental scenarios for Java source code. They perform pervasive modifications of source code using the ARTIFICE tool [97] and ByteCode obfuscation using the ProGuard tool\(^1\), which optimizes Java ByteCode and provides reverse engineering by obfuscating the names of classes, fields and methods. The modification includes changes in layout or renaming of identifiers, changes that affect the code globally. Local modification of code is also performed. They use compilation and decompilation for transformation (obfuscation) and normalization, respectively.

Despite having a plethora of tools and methods for detecting code obfuscation, their effectiveness is unclear. The reason is that they analyze the structure of the code and pay scant importance to the semantics or meaning of the code. As a result, structural variations of the code remain undetectable using the above methods. Because of the resemblance of the problem to semantic code clone detection, we model it as a code clone detection

\(^1\)https://www.guardsquare.com/en/proguard
problem. We propose a single detection framework for both semantic Java code clones and obfuscated Java code using machine learning.

**An Integrated Detection Framework**

A straightforward approach to determine if two fragments of code are semantically similar without necessarily being syntactically similar may proceed as follows: trim and normalize the two fragments as discussed in Machine Learning for Pairwise Clone Detection Section, obtain the formal semantics of the two using a method alluded to earlier; and, compare the formal semantic representations using Equation 4.3 section. However, tools to obtain formal semantics are not readily available. In addition, formal semantic representations are strings themselves, requiring additional string comparisons. It is also unclear that formal semantic representations will add substantially to efficient and effective code clone or obfuscated code detection. Thus, it may be appropriate to investigate if other approaches may work well in detecting if two code fragments are semantically similar with each other, and additionally if they are obfuscated.

Code clone or obfuscated code detection has been treated as a pairwise similarity analysis problem, where two pieces of code are semantic clones or one is an obfuscated version of the other if a given piece of code is semantically similar to the given reference code. However, machine learning usually considers individual samples for training and predicts class labels. Instead of comparing the structural and meaning representations (which may be long and/or difficult-to-obtain strings themselves) directly, to compare if two codes are syntactically or semantically similar, we can extract relevant characteristics of the code
fragments by looking at selected portions of them or other associated structures. Such characteristics are usually called features in the machine learning literature. To apply machine learning to pairwise clone detection, we use features of both the reference and target code fragments.

The similarity between two code fragments is measured by computing similarity between the two feature based representations. The relevant features for a pair of code fragment can come from many sources. One such source is ByteCode Dependency Graph (BDG) representation. Java ByteCode representation is less ambiguous than high-level source code. In our work, we use broadly two categories of code fragments features, ByteCode or low level features and source code or high level features. Type of features we use in our work is enlisted below.

- **Low Level Features**
  - ByteCode (BC) features
  - ByteCode Dependency Graph (BDG) features

- **High Level Features**
  - Traditional features
  - Abstract Syntax Tree (AST) features
  - Program Dependency Graph (PDG) features

Next, we discuss in details the code fragment features we use for semantic clone or obfuscated code detection.
Java ByteCode : Low Level Features

Java source programs are compiled into a portable binary format called ByteCode. The ByteCode is an intermediate program between Java source code and machine code. Java ByteCode is a sequence of instructions for the virtual machine to execute basic functionalities such as conditions and loops. Each ByteCode contains one or more opcodes. Java Virtual Machine takes the ByteCode and converts it into machine code. When a Java virtual machine (JVM) loads a class file, it is executed by an interpreter. This file contains a stream of ByteCode instructions for each method in the class. ByteCodes are low-level representation for Java programs and hence are likely to be effective for representing the semantics of a program.

We attempt to represent the meaning of a program by elucidating interdependency relationships among different ByteCode constructs. We represent such dependencies as a graph called the ByteCode Dependency Graph (BDG). An illustration of a BDG construction scheme is depicted in Figure 6.1. BDGs represent both data and control dependencies for each operation in the ByteCode. We create a BDG from the ByteCode and extract features from the graph. The BDG features are the semantic or meaning features. We extract control dependency features by reading the .class file sequentially and by tracking down all the instructions that may cause conditional or unconditional branching of the control flow of the code sequence. We consider three types of control instructions, which are listed in the Table 6.1. We find the frequencies of various data and control dependency relationships among different instructions. We consider a total of 23 constructs and 85 relationships between them and use them as our BDG.
Table 6.1: ByteCode conditional statements

<table>
<thead>
<tr>
<th>Control</th>
<th>Instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unconditional Branch</td>
<td>goto, goto_w</td>
</tr>
<tr>
<td>Conditional Branch</td>
<td>ifeq, iflt, ifle, ifgt, ifge, ifnull, ifnonnull, if_icmpeq, if_icmpgt, if_icmpge, if_acmpeq, if_icmplt, if_icmple, if_icmpne, if_acmpne</td>
</tr>
<tr>
<td>Compound Cond. Branch</td>
<td>tableswitch, lookupswitch</td>
</tr>
<tr>
<td>Comparisons</td>
<td>lcmp, fcmpg, fcmlp, dcmpg, dcmlp</td>
</tr>
</tbody>
</table>

Table 6.2: Categorization of ByteCode instructions

<table>
<thead>
<tr>
<th>Category</th>
<th>Instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load</td>
<td>aload, dload, fload, iload, lload</td>
</tr>
<tr>
<td>Store</td>
<td>astore, dstore, fstore, istore, lstore</td>
</tr>
<tr>
<td>const</td>
<td>aconst, icong, lconst, fconst, lconst</td>
</tr>
<tr>
<td>Arithmetic</td>
<td>iadd, dadd, fadd, iadd, isub, dsub, fsub, lsub, imul, dmul, fnmul, lmul, idiv, ddiv, fdv, ldiv, irem, drem, frem, lrem</td>
</tr>
<tr>
<td>Type Conversion</td>
<td>i2l, i2f, i2d, i2f, l2f, l2d, f2i, f2l, f2d, d2i, d2l, i2b, i2c, i2s</td>
</tr>
</tbody>
</table>

Figure 6.1: A BDG showing control and data dependency among instructions.
We feel that similarity between two blocks of ByteCode can be used as a measure of similarity between the semantics of a pair of original source code fragments. We preprocess the source code by trimming and normalizing as discussed earlier. We first extract syntactical features from the program ByteCode. Frequency of occurrence of various ByteCode instructions such as load, store, add, and sub, are used as features, what we call ByteCode features. When parsing the .class file, we ignore certain ByteCode entities like statement numbers. Such low information entities are unlikely to contribute to the meaning of a program and hence we remove them. We classify the instructions into several categories for our ease of computing the features and dependency relationships. The categories are listed in Table 6.2.

We extract both ByteCode (BC) and BDG features from the BDG itself. Algorithm 6 shows the steps in extracting such features from a BDG. It takes Java code as input and generates the ByteCode as a .class file using the Javac compiler. $L_{BC}$ and $L_{BDG}$ are the vectors of pre-specified BC and BDG attributes, respectively. The algorithm counts the frequencies of the target attributes for both BC and BDG. In case of BC features, $\text{MatchToken}$ matches each pre-specified BC attribute and increases the count of the frequency of the target feature in the $L_{BC}$ vector. To extract BDG features, the method $\text{DependencyFreq}$ checks for all possible control dependency relationships that persist in the ByteCode. Similar to $\text{MatchToken}$, it increases the count if the specified relationship, given as BDG attribute is encountered during each iteration. It returns the frequency vector and stores it as $L_{BDG}$. Frequencies of BC and BDG attributes are finally stored into a vector $F$ as features of a given Java program. Please refer to Tables 6.1, 6.2 and Appendix B materials for the details of the features we extract as BC and BDG features.
Algorithm 6 ByteCode & BDG Feature Extraction

1: **INPUT**: $C$ // Java Source Code
2: **OUTPUT**: $F = \{f_{BC1}, \cdots, f_{BCN}, f_{BDG1}, \cdots, f_{BDGM}\}$ // Set of $N$ BC and $M$ BDG features
3: **Steps**:
4: $L_{BC} = \{B_1 \cdots B_N\}$ // List of $N$ BC attributes
5: $L_{BDG} = \{D_1 \cdots D_M\}$ // List of $M$ BDG attributes
6: $T \leftarrow$ javac($C$) //invoking Javac compiler
7: $V \leftarrow$ Tokenize($T$) //Read line by line the .class file and store the stream sequence of instructions in a vector $V$
8: for $i = 1 \cdots |L_{BC}|$ do
9: for $j = 1 \cdots |V|$ do
10: if MatchToken($B_i$, $V_j$) then
11: $f_{BC_i} = f_{BC_i} + 1$
12: end if
13: end for
14: $F = F \cup f_{BC_i}$
15: end for
16: //Counting frequency of BDG features
17: for $i = 1 \cdots |L_{BDG}|$ do
18: $f_{BDG_i} \leftarrow$ DependencyFreq($D_i$, $V$)
19: $F = F \cup f_{BDG_i}$
20: end for
21: return $F$

Source Code Features

Besides using low level ByteCode for capturing the semantics of a Java code snippet we also extract semantic as well as syntactic features directly from the high level of the source code. The combination of both low and high label features may represent the semantics and syntax more accurately, hence, helping in better matching of two target programs.

Syntactic similarity between two code blocks is also likely to impact upon the similarity in meaning, and hence we also parse the code fragments into their structural components in terms of Abstract Syntax Tree (AST) [10]. Each node of the tree represents a
programming construct occurring in the given source code. Leaf nodes of the tree contain variables. Unlike a majority of published clone detection methods that compare the two syntactic trees directly, we extract certain characteristics or features from the ASTs. Figure 4.1 shows an example AST created by the AST Generator software we use. We traverse the AST in post-order manner and extract only non-leaf nodes containing programming constructs such as Variable Declaration Statements (VDS), While Statements, Cast Expressions, Class Instances, and Method Invocations. We represent the frequencies of these programming constructs as AST features in a vector.

We also extract source code control dependency features from Program Control Dependency Graph (PDG) [32]. The PDG features are a type of semantic features. PDGs make explicit both the data and control dependence for each operation in a program. Data dependencies represent the relevant data flow relationships within a program. Control dependencies represent the essential control flow relationships. A sample PDG derived from a fragment of code is given in Figure 4.2. Edges represent the order of execution of program nodes. Nodes represent the lines where the corresponding elements are located in the program. Horwitz et al. [43] show that PDGs can be used as “adequate” representations of programs and prove that if the PDGs of two graphs are isomorphic, they are strongly equivalent, i.e., they are “programs with the same behavior.” We extract features, instead of directly matching the two PDG graphs for clone detection. Examples of such features are the number of Assignments that come after Declarations, obtained by counting the occurrence of the assignments which are dependent on declarations; the number of Declarations

\[\text{http://www.eclipse.org/jdt/}\]
coming after Control (e.g., i < count, for, while, if, switch etc.), the number of times nested
iterations occur; the number of times nested selections occur, and so on.

In addition to all of the features discussed above, we extract basic source code char-
acteristics, which we call Traditional Features. They include number of Lines of Code
(LOC), numbers of keywords, variables, assignments, conditional statements and iteration
statements [52] used in a given piece of source code.

**Fusion of Code Features**

We combine feature vectors (Equation 6.3) extracted from a pair of target and ref-
erence code fragment to create the training dataset, which is similar to Fusion of Block
Features Section.

\[
[C^T_i] \approx <f^l_{i1}, \ldots f^l_{ik_b} | f^h_{i1}, \ldots f^h_{ik_d} | f^t_{i1}, \ldots f^t_{ik_a} | f^p_{i1}, \ldots f^p_{ik_p}>. \tag{6.3}
\]

In this equation, we denote the different categories of features with different super-
scripts: \(l\) for low level or ByteCode related features and \(h\) for high level source code fea-
tures. We denote the different types of features with different superscripts: \(b\): ByteCode, \(d\):
BDG, \(t\): traditional, \(a\): AST and \(p\): PDG. Features are separated into five different groups
with vertical lines, for clear separation distinction.

We fuse the sequence of features from the two different code fragments. Although
there are five types of features covering both low and high level features in the description
of a code fragment, to simplify the notation, we rewrite Equations 6.4 and 6.5, without
distinguishing among the feature types, as:
\[ [C_i^n] \approx \text{features}(C_i) = <f_{i1}, \cdots f_{ik}>. \]  

(6.4)

where \( k = k_b + k_d + k_t + k_a + k_p \). Similarly,

\[ [C_j^n] \approx \text{features}(C_j) = <f_{j1}, \cdots f_{jk}>. \]  

(6.5)

We use known pairs of cloned or obfuscated code fragment for feature extraction and label the class of the feature vector as true clone or with the obfuscation type.

Given two code fragments \( C_i \) and \( C_j \), and the corresponding class label \( D \) for the code fragments, the combined feature vector, \( \text{features}(<C_i, C_j>) \) can now be represented as a fused feature vector. We fuse the two vectors in three different ways as discussed in Fusion of Block Features Section.

**Code Similarity as a Feature**

We use the text similarity score between a pair of target code fragments as one of the features. We compute text similarity between two target Java source code fragments \( C_i \) and \( C_j \), and also the text similarity between their corresponding ByteCodes \( B_i \) and \( B_j \) respectively. We tokenize each source code fragment and count frequencies of the tokens. We calculate cosine similarity [15] using the frequency vectors as follows,

\[ S_1(C_i, C_j) = \cos(\theta) = \frac{\sum_{i,j} c_i \times c_j}{\sqrt{\sum_i c_i^2} \times \sqrt{\sum_j c_j^2}} \]  

(6.6)
where, \( c_i \) and \( c_j \) are the components of the frequency vectors for \( C_i \) and \( C_j \), respectively.

Similarly, we calculate the similarity between two Java ByteCode fragments, \( S_2(B_i, B_j) \), using the above equation.

Finally, a feature vector for a given pair of code fragments for training can be represented as follows.

\[
\text{features}(\langle C_i, C_j \rangle) = <f_{ij1}, \cdots, f_{ijk}, S_1(C_i, C_j), S_2(B_i, B_j), D>.
\] (6.7)

We use a total of 258 features to represent a pair of code blocks. The distribution of features categories we use to train a classification model is shown in Figure 6.2.

**A New Code Obfuscation and Clone Detection Scheme**

We use our machine learning framework both for detecting code clones as well as obfuscated code. Like any other machine learning framework, our scheme also has two phases, training and testing. In training, we use labeled pairs of cloned code or known obfuscated code from a given corpus. We perform pre-processing steps, including trimming and normalization. Next, we compile the code blocks to Java ByteCode classes or files. Then, we generate both low level and high level features from the given pair of code in terms of BC, BDG, AST and PDG features and fuse feature vectors of two target code...
blocks using one of the Equations 4.10, 4.11 or 4.12. We compute similarity between the pair of code blocks using cosine similarity and append them into the fused feature vector. We label the feature vector with a class label based on clone type or whether it is obfuscated code or not. We use a binary classifier for detection of semantic clones or possible obfuscated code. Accordingly, we mark it as Y on N to indicate semantic clone or obfuscated code.

All the above steps are iterated for all possible pairs of code fragments to create a training dataset for the classification model. To identify possible clone or obfuscation in a pair of unlabeled code blocks, we perform the same sequence of steps to create a fused feature vector of the two given blocks and pass it through the classifier for prediction of the possible clone type or to determine if one is an obfuscated version of the other. Figure 6.3 demonstrates the work-flow of our approach. We also explore the use of a classifier ensemble using the majority voting approach [26] with a hope of achieving better detection rate.

**Experimental Evaluation**

We evaluate and establish the superiority of our proposed detection framework. In our experiments, we use only examples of Java source code as a corpus for training and testing. However, this model is general in nature and can be extended easily to any other high level programming language. Our primary goal is to improve accuracy of detection of semantic clones as well as obfuscated code. We use an ensemble of selected classifiers and
Figure 6.3: Workflow of the proposed dual detection framework.
compare the effectiveness of the proposed framework with the state-of-the-art clone and obfuscated code detection methods.

**Datasets**

We use a number of datasets for both Java code clones and obfuscated code separately. We discuss them in brief below.

**Clone Dataset**

We use six Java code clone and three obfuscated code datasets for training and testing. Details of the datasets are given in Tables 5.1 and 6.3. A majority of existing clone datasets used in prior papers are incomplete in nature. They avoid labeling semantic code clones. The publicly available datasets are *eclipse-ant, netbeans-javadoc, j2sdk14.0-javax-swing, eclipse-jdtcore, EIRC* and *Suple.*
Obfuscated Code Dataset

We generate examples of obfuscated code using available obfuscation tools. At first we use five Java classes namely *InfixConverter, SqrtAlgorithm, Hanoi, EightQueens*, and *MagicSquare* to generate Java obfuscated code and name the entire set of classes ObsCode. All of these classes are less than 200 LOC. Each class of Java code is obfuscated using Artifce [97]. Then, the original and obfuscated files are compiled to ByteCode. Both ByteCode files are obfuscated further using ProGuard to create stronger obfuscation. After that, all four ByteCode files are decompiled using either Krakatau³ or Procyon⁴ giving back eight additional obfuscated source code files [84]. We generate nine pervasively modified version of each original source code, resulting in a total 50 of files for the dataset [84].

We select PacMan game⁵ as our second subject system [97]. It contains 21 files with 2400 lines of code. The classes are further transformed using renaming, contraction, expansion, and loop transformations [97].

We also select supplementary source programs available in the textbook called *Algorithms*⁶ containing 149 Java source files and generate the obfuscated codes. We generate obfuscation files from the above source files using the approach illustrated in Figure 6.5. Each class of Java code is obfuscated using ProGuard. Then, the original and obfuscated files are compiled to ByteCode. Both ByteCode files are obfuscated once again using ProGuard. After that, all ByteCode files are decompiled using Java decompiler giving back

³https://bitbucket.org/mstrobel/procyon/wiki/Java
⁴https://github.com/Storyyeller/Krakatau
⁵https://code.google.com/p/pacman-rkant/
⁶http://algs4.cs.princeton.edu/home/
obfuscated source code files. We obtain 785 Java ByteCode files for the dataset. A summary of the dataset is given in Table 6.3.

**Ensemble Classification Model**

We train and test our model using an ensemble approach using majority voting [26] between ten classifiers. We include classification decision from Naïve Bayes [48], LibLinear SVM [30], Instance Based Learner (IBK) [3], Bagging [12], Logit Boost [33], Random Committee [110], Random Subspace [42], Rotation Forest [89], J48 [96], and Random Forest [13] classifiers, and ensemble them based on majority decisions to obtain the final class label.
Experimental Results

We generate extensive results to assess the robustness of our proposed model in detecting semantic clones and obfuscated code. We experiment with a varying number of features and with different feature fusions schemes to show that our features are able to achieve a high detection accuracy.

Experimenting with varying feature fusion methods

We assess the importance of combining Traditional, AST, PDG, ByteCode and BDG features and report the results produced by the proposed framework on both clone and obfuscated datasets in Figures 6.7 and 6.8, respectively. Results produced by the ensemble classifier with varying feature fusion methods, show that the performance of the ensemble classifier improves substantially as we combine both syntactic and semantic features to detect clones. Interestingly, the performance of the classifier using semantic features is consistent irrespective of feature types and fusion methods. We also observe that distance
Figure 6.7: Performance of detection framework on clone datasets with different feature combinations.

Figure 6.8: Effectiveness of the framework on detecting obfuscated code using feature fusions.

and multiplicative combinations produce better results than linear combination for all sizes of data.

In case of obfuscated datasets, the results reported in Figure 6.8 show that we achieve 100% accuracy for the first two datasets irrespective of the feature fusion method used. However, linear fusion gives better results in comparison to other methods.
Experimenting with selected features

We perform two different kinds of experiments with varying numbers of features, selecting equal numbers of features from each feature type (Traditional, AST and PDG, ByteCode, and BDG) and using a feature selection method (Figure 6.10). The intention behind such experiments is to show the significance of our features in achieving better accuracy, and that it is not by chance. The growing learning curve (Figure 6.9) clearly indicates that the detection accuracy improves with the increase in the numbers of features.

In another experiment, instead of selecting features by category, we use a random forests based feature selection algorithm, namely Mean Decrease Impurity (MDI) [69] for selecting feature vectors after applying different fusion methods. Random forests provide an easy way to assess importance of features based on majority decision using an ensemble of randomized trees. For each experiment, we use different numbers of the feature sets, ranked by the feature selection algorithm. Figure 6.10 and 6.11 report the results on clone and obfuscated datasets. Similar to the learning curve based on randomly selected
Figure 6.10: Performance of ensemble approach on clone dataset after feature selection using *MDI*.

![Graph showing accuracy vs number of features](image)

Figure 6.11: Effectiveness of ensemble classifier for detecting obfuscated code after feature selection using *MDI*.

![Graph showing accuracy vs number of features](image)

feature sets, growth in the size of selected feature sets shows how a growing trend in the performance. This further establishes the fact that our features are crucial in deriving high accuracy detection rates, especially in detecting obfuscated code.
Performance comparison

We compare the performance of our method with state-of-the-art clone detectors and contemporary obfuscated code detection tools.

Comparison of Clone Detectors

We compare the performance of our framework with contemporary clone detection methods, using reported results on eclipse-ant, netbeans-javadoc, j2sdk14.0-javax-swing, eclipse-jdtcore, EIRC and Suple datasets. Prior research reports a range of F-scores for detecting Type I, Type II and III clones [76] and [44]. This is because available clone datasets lack Type IV examples. Moreover, a majority of the detection methods are incapable of detecting semantic clones. Hence, we conduct analysis of the clone detectors for their effectiveness in detecting Type I, II and III clones. We compare our framework’s performance with the maximum value of the reported range by other authors. Figure 6.12 shows comparison of our results with the state-of-the-art detectors in terms of F-score. Results clearly establish that our method is superior in detecting all type of clones.

For reliable future error prediction, when we build a model, we need to evaluate it to compare it with other models. We need to evaluate the model independent datasets that are different from datasets we have used for building the model. We integrate j2sdk14.0-javax-swing and eclipse-jdtcore systems and use the integrated dataset as a training set, build the model and estimate the parameters of the model during the learning phase as in Figure 6.4. Then, we use a different and completely new dataset for model evaluation.
Figure 6.12: Prediction effectiveness of proposed framework in comparison to state-of-the-art clone detectors in terms of F-Score.

(a) Detection rates for Type-I and II clones

We build our model using distance combination and Traditional, AST, PDG, BC and BDG features and report accuracy results in Figure 6.4. The performance of the ensemble classifier using all features, which are Traditional, AST, PDG, BC and BDG features, is better than the performance of the ensemble classifier using only AST and PDG features.
Performance of Obfuscated Code Detectors

We use previously published performance scores [84] in terms of Precision, Recall and F1-score values. We present the maximum value of the range reported by other authors, to give benefit of the doubt to our competitors. Figure 6.13 shows a comparison of our results with different obfuscated code detectors based on recall, precision and F1-Score on the ObsCode dataset. It is evident that our method performs better than all other methods in detecting obfuscated code. Our method is the winner with the highest F1-Score (100%) in all cases of detecting obfuscated code, using three different datasets, ObsCode, ObsCode*(karkatau), and ObsCode* (procyon). ObsCode*(karkatau), and ObsCode* (procyon) are the variations of our ObsCode dataset, created using three different obfuscation tools, Artifice, ProGuard, and Decompilers.

In Figure 6.14, we also compare our method with three different obfuscation code detection tools, selecting each tool based on the particular detection method they use. In terms of accuracy, our method is the best compared to the other four methods, which are Text-based (JPLAG(v.2.2.1)), Token-based (JPLAG(v.2.2.1)), AST-base (CloneDigger), and PDG-based (Scorpio). Our approach achieves 100% accuracy in most of the cases. When we compare our method with other methods for all obfuscations types, viz, contraction, expansion, loop transformation and renaming, our model is the winner with the highest accuracy (98.4%) followed by Token-based (JPLAG(v.2.2.1)) (91.6%), Text-based (JPLAG(v.2.2.1)), PDG-based (Scorpio) (48.8%), and AST-based (CloneDigger) (38.5%), respectively.
Figure 6.13: Effectiveness of various obfuscated code detection tools on ObsCode dataset.

(a) Recall

(b) Precision

(c) F1-Score
Figure 6.14: Tool performance comparison on the PacMan data in terms of accuracy, in which the original program is compared to obfuscated programs for each clone detection technique. The obfuscation methods are abbreviated as follows: C–contraction, E–expansion, L–loop transformation, R–renaming [97].

We build a model for detecting code obfuscation using three ways. 1) The model is built using the combined obsCode+Algorithms dataset and evaluated using the Pac-Man dataset. 2) The model is built using the combined obsCode+PacMan dataset and evaluated using the Algorithms dataset. 3) The model is built using the combined Pac-Man+Algorithms dataset and evaluated using the ObsCode dataset, as shown in Figure 6.6. In terms of accuracy, our obfuscation code detection model produces better results even when the model is built with a dataset and tested on another.

Chapter Summary

The semantics of a program written by an anonymous programmer is difficult to characterize, especially for detecting software clones and obfuscated programs. A number of methods and software tools are available for the purpose of detecting code clones or
obfuscated code. We propose a novel integrated framework for detecting both Java code clones and Java obfuscated code. We capture the semantics of program code using low and high level program features derived from ByteCode, the data and conditional dependency within the ByteCode, AST and PDG. We perform an extensive set of experiments to show that our framework is able to detect both code clone and obfuscated code equally well. The detailed results we present in this chapter and in Appendix B clearly establish that the machine learning approach we use with the carefully obtained features as the current best method for simultaneously detecting all four types of clones as well as obfuscated code. The current framework is limited to only on Java code. We are exploring ways to make the framework more general in nature, with an eye to its high commercial importance.
CHAPTER VII

CONCLUSION AND FUTURE WORK

Conclusion

In this thesis, we applied machine learning as a tool for detecting software code clones. We started with a comprehensive study on state-of-the-art clone detection tools and compared their performance.

We used normalized blocks, followed by grouping and hashing to detect the Type-I and Type-II clones. We used two similarity measures to detect Type-III clones.

We also proposed a machine learning approach for detecting and extracting semantic as well as synthetic clones. A novel features extracted from Abstract Syntax Trees (ASTs) and Program Dependency Graphs (PDGs) were used to detect various types of Java code clones.

Majority of the publicly available datasets lack Type-IV labeling, without which it is difficult to run and validate any semantic clone detection methods. We presented a new framework for labeling semantic code clones using Java ByteCode similarity to label pub-

Finally, we presented a potential application of our proposed clone detection framework for detecting obfuscated code. An integrated framework for detecting both Java code clones and obfuscated code was presented in the thesis. We used high-level source code features from Abstract Syntax Tree (AST) and Program Dependency Graph (PDG) of the code and low level code features from Java ByteCode and Byte Dependency Graph (BDG) to detect code clones and obfuscation. We used an ensemble of state-of-the-art classification models to evaluate the effectiveness of our proposed idea.

All the proposed methods were evaluated experimentally and compared with state-of-the-art methods and tools using various publicly available clone and obfuscated datasets. Results show that our proposed methods are superior in performance in detecting both code clones and obfuscated codes.

**Future Work**

The work presented in this thesis can be extended further. Below we list some ideas for future work.

- Our current work is restricted to only Java code. A future endeavor for developing a general framework covering all major programming languages will have better commercial value.

- Semi-supervised learning has focused on designing algorithms to effectively exploit unlabeled data. It uses a small amount of labeled data to help label unlabeled data
and reduce the cost associated with labeling. We plan to construct clone references for Type-IV clones using semi-supervised learning algorithms that can be trained at less expense to identify semantic clones in different programing languages for helping researchers evaluate future code clone detection techniques.

- Convolutional Neural Network (CNN or ConvNet) [58] is a type of feed-forward artificial neural network with many layers. We plan to explore the use of the ConvNet model to detect Java code clones and obfuscated code using our novel high-level source features from Abstract Syntax Tree (AST) and Program Dependency Graph (PDG) of the code, and low level features from Java ByteCode and Byte Dependency Graph (BDG).

- A set of novel features can be extracted from assembly language (low level) generated from source code. We plan to extend our work to further improve detecting types of clones using assembly language features. We intend to use novel assembly language features along with program dependency graph (PDG) and abstract syntax tree (AST) features, and ByteCode dependency graph (BDG). BDG, PDG, and assembly language are alternative representations of the semantics or meaning of a Java program. AST captures the structural aspects of a program.

- We also plan to propose a novel machine learning framework for both automated detection of all four types of clones, obfuscated code and malicious executables using features extracted from assembly instruction sequences, and Dynamic Link Library (DLL) function calls, extracted from binary executables, disassembled executables, and executable headers, respectively, using code block pairs as training examples. We will introduce a formal way to model code clone detection, obfuscation code and malicious
executables problems, and use state-of-the-art classification models to assess the prediction performance of our scheme.
REFERENCES


[18] Mihai Christodorescu, Somesh Jha, Sanjit Seshia, Dawn Song, and Randal E


[38] Jiawei Han and Micheline Kamber. Data mining: Concepts and techniques, university of illinois at urbana-champaign, 2006.


[68] Chao Liu, Chen Chen, Jiawei Han, and Philip S Yu. Gplag: detection of software plagiarism by program dependence graph analysis. In Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pages 872–881. ACM, 2006.


[70] D Jurafsky JH Martin and D Jurafsky. Speech and language processing. an introduction to natural language processing, computational linguistics, and speech recognition, 2005.

[71] Jean Mayrand, Claude Leblanc, and Ettore M Merlo. Experiment on the auto-


[87] Konrad Rieck, Philipp Trinius, Carsten Willems, and Thorsten Holz. Automatic


[111] Xifeng Yan, Jiawei Han, and Ramin Afshar. Clospan: Mining closed sequential patterns in large datasets. In Proceedings of the 2003 SIAM International Conference on Data Mining, pages 166–177, 2003.


## Appendix A

### FEATURES AND RESULTS OF CHAPTER IV

#### Traditional Features

<table>
<thead>
<tr>
<th>Category</th>
<th>Features</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Traditional</td>
<td>No. of Lines</td>
<td>Counts lines of a method blocks.</td>
</tr>
<tr>
<td></td>
<td>No. of Assignments</td>
<td>Counts Assignments in method blocks.</td>
</tr>
<tr>
<td></td>
<td>No. of Selection Statements</td>
<td>Counts selection or condition statements in method blocks.</td>
</tr>
<tr>
<td></td>
<td>No. of Iteration Statements</td>
<td>Counts iterations or loop statements in method blocks.</td>
</tr>
<tr>
<td></td>
<td>No. of Synchronized Statements</td>
<td>Counts &quot;synchronized ( Expression ) blocks.&quot; in method blocks.</td>
</tr>
<tr>
<td></td>
<td>No. of Return Statements</td>
<td>Counts &quot;return [ Expression ] ;&quot; in method blocks.</td>
</tr>
<tr>
<td></td>
<td>No. of SwitchCase Statements</td>
<td>Counts &quot;case Expression : default :&quot; in method blocks.</td>
</tr>
<tr>
<td></td>
<td>No. of Try Statements</td>
<td>Counts &quot;try { ( Resources ) } blocks. [ CatchClause ] { finally blocks } ;&quot; in method blocks.</td>
</tr>
<tr>
<td></td>
<td>No. of Single Variable Declaration</td>
<td>Counts &quot;{ ExtendedModifier } Type [Annotation] { ... } Identifier [Dimension] [ = Expression ] ;&quot; in method blocks.</td>
</tr>
<tr>
<td></td>
<td>No. of Variable Declaration</td>
<td>Counts &quot;SingleVariableDeclaration and VariableDeclarationFragment ;&quot; in method blocks.</td>
</tr>
<tr>
<td></td>
<td>No. of Variable Declaration</td>
<td>Counts &quot;{ ExtendedModifier } Type VariableDeclarationFragment { , VariableDeclarationFragment } ;&quot; in method blocks.</td>
</tr>
<tr>
<td></td>
<td>No. of Expression Statements</td>
<td>Counts &quot;StatementExpression ;&quot; in method blocks.</td>
</tr>
<tr>
<td></td>
<td>No. of Type Declaration</td>
<td>Counts &quot;TypeDeclaration and EnumDeclaration&quot; in method blocks.</td>
</tr>
<tr>
<td></td>
<td>No. of Type Declaration</td>
<td>Counts &quot;{ ExtendedModifier } Identifier [ extends Type</td>
</tr>
</tbody>
</table>

Table 1.1: *Traditional* features
Table 1.2: AST features

<table>
<thead>
<tr>
<th>Category</th>
<th>Features</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>No. of Class Instance Cre-</td>
<td>Count `&quot;new PrimitiveType [ ] Expression . [ ] ] [ ] new TypeName [ ] Type [ ] [ ] [ ] Type . Type [ ] [ ] [ ] ArrayInitializer [ ] ] new Instance [ ] Class [ ] Type [ ] [ ] ] ] in method blocks.</td>
<td></td>
</tr>
<tr>
<td>ations</td>
<td>No. of Array Creation</td>
<td>No. of Instance Creation</td>
</tr>
<tr>
<td>cations</td>
<td>No. of Array Types</td>
<td>No. of Array Types</td>
</tr>
<tr>
<td>zers</td>
<td>No. of Array Accesses</td>
<td>No. of Array Accesses</td>
</tr>
<tr>
<td>No. of Continue Statements</td>
<td>Count `&quot;continue [ Identifier . ] in method blocks.</td>
<td>Count `&quot;continue [ Identifier . ] in method blocks.</td>
</tr>
<tr>
<td>tions</td>
<td>No. of This Expressions</td>
<td>No. of This Expressions</td>
</tr>
</tbody>
</table>
## PDG Features

<table>
<thead>
<tr>
<th>Category</th>
<th>Features</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PDG</td>
<td>No. of AC</td>
<td>Counts Assignment (\rightarrow) Control in method blocks.</td>
</tr>
<tr>
<td></td>
<td>No. of AR</td>
<td>Counts Assignment (\rightarrow) Return in method blocks.</td>
</tr>
<tr>
<td></td>
<td>No. of EC</td>
<td>Counts Expression (\rightarrow) Control in method blocks.</td>
</tr>
<tr>
<td></td>
<td>No. of DA</td>
<td>Counts Control Assignment (\rightarrow) Assignment in method blocks.</td>
</tr>
<tr>
<td></td>
<td>No. of CA</td>
<td>Counts Control Assignment (\rightarrow) Declaration in method blocks.</td>
</tr>
<tr>
<td></td>
<td>No. of ACD</td>
<td>Counts Control Assignment (\rightarrow) Declaration (\rightarrow) Assignment in method blocks.</td>
</tr>
<tr>
<td></td>
<td>No. of CW</td>
<td>Counts Control While statement (\rightarrow) Assignment in method blocks.</td>
</tr>
<tr>
<td></td>
<td>No. of CF</td>
<td>Counts Control For statement (\rightarrow) Assignment in method blocks.</td>
</tr>
<tr>
<td></td>
<td>No. of CD</td>
<td>Counts Control Declaration (\rightarrow) Assignment in method blocks.</td>
</tr>
<tr>
<td></td>
<td>No. of CIA</td>
<td>Counts Control If statement (\rightarrow) Assignment in method blocks.</td>
</tr>
<tr>
<td></td>
<td>No. of CSA</td>
<td>Counts Control Selection (\rightarrow) Assignment in method blocks.</td>
</tr>
<tr>
<td></td>
<td>No. of CD</td>
<td>Counts Control Declaration in method blocks.</td>
</tr>
<tr>
<td></td>
<td>No. of AD</td>
<td>Counts Assignment (\rightarrow) Declaration in method blocks.</td>
</tr>
<tr>
<td></td>
<td>No. of CW</td>
<td>Counts Control While statement (\rightarrow) Declaration in method blocks.</td>
</tr>
<tr>
<td></td>
<td>No. of CF</td>
<td>Counts Control For statement (\rightarrow) Declaration in method blocks.</td>
</tr>
<tr>
<td></td>
<td>No. of CD</td>
<td>Counts Control Declaration (\rightarrow) Declaration in method blocks.</td>
</tr>
<tr>
<td></td>
<td>No. of CIA</td>
<td>Counts Control If statement (\rightarrow) Declaration in method blocks.</td>
</tr>
<tr>
<td></td>
<td>No. of CSA</td>
<td>Counts Control Selection (\rightarrow) Declaration in method blocks.</td>
</tr>
<tr>
<td></td>
<td>No. of Iteration</td>
<td>Counts Iteration statement (\rightarrow) Iteration statement in method blocks.</td>
</tr>
<tr>
<td></td>
<td>No. of Selection</td>
<td>Counts Selection statement (\rightarrow) Selection statement in method blocks.</td>
</tr>
<tr>
<td></td>
<td>No. of NestedLoop</td>
<td>Counts Iteration statement (\rightarrow) Return statement in method blocks.</td>
</tr>
<tr>
<td></td>
<td>No. of NestedSelection</td>
<td>Counts Selection statement (\rightarrow) Return statement in method blocks.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Nested Selection</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Nested Loop</td>
</tr>
</tbody>
</table>
## Results of Combinations of Pair Instances Vectors

### Table 1.4: Results of distance and multiplicative combinations of pair instances vectors

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Features</th>
<th>Type1/2</th>
<th>Accuracy</th>
<th>f1-score</th>
<th>Precision</th>
<th>Recall</th>
<th>Measure</th>
</tr>
</thead>
<tbody>
<tr>
<td>Xgboost</td>
<td>Features</td>
<td>True</td>
<td>0.77</td>
<td>0.77</td>
<td>0.77</td>
<td>0.77</td>
<td>0.77</td>
</tr>
<tr>
<td>Random Forest</td>
<td>Features</td>
<td>True</td>
<td>0.77</td>
<td>0.77</td>
<td>0.77</td>
<td>0.77</td>
<td>0.77</td>
</tr>
<tr>
<td>Traditional Features (14)</td>
<td>True</td>
<td>0.77</td>
<td>0.77</td>
<td>0.77</td>
<td>0.77</td>
<td>0.77</td>
<td>0.77</td>
</tr>
<tr>
<td>Synthetic Features (43)</td>
<td>True</td>
<td>0.77</td>
<td>0.77</td>
<td>0.77</td>
<td>0.77</td>
<td>0.77</td>
<td>0.77</td>
</tr>
<tr>
<td>All Features (100)</td>
<td>True</td>
<td>0.77</td>
<td>0.77</td>
<td>0.77</td>
<td>0.77</td>
<td>0.77</td>
<td>0.77</td>
</tr>
<tr>
<td>False</td>
<td>0.77</td>
<td>0.77</td>
<td>0.77</td>
<td>0.77</td>
<td>0.77</td>
<td>0.77</td>
<td>0.77</td>
</tr>
</tbody>
</table>

### Notes
- The table provides results for different combinations of features and measures.
- Accuracy, f1-score, Precision, and Recall are used to evaluate the performance of the algorithms.
- The results are presented for three different types of algorithms: Xgboost, Random Forest, and Traditional Features.
- The features are divided into two categories: Traditional Features (14) and Synthetic Features (43), with an additional All Features (100) category.
- The performance metrics are shown for both True and False conditions.

### Further Analysis
- The Xgboost algorithm shows similar performance across all feature types.
- Random Forest also provides consistent results, slightly lower than Xgboost in some cases.
- Traditional Features (14) show comparable accuracy to Synthetic Features (43) when True, but slightly lower when False.
- The All Features (100) category consistently outperforms the others in accuracy, f1-score, Precision, and Recall across both True and False conditions.

### Conclusion
- The use of an All Features (100) combination leads to the best performance across all metrics for both True and False conditions.
- Further experiments may be needed to determine the optimal feature set for specific applications.
## Results of BigCloneBench Recall, Precision and F-Measure measurements

Table 1.5: BigCloneBench Recall, Precision and F-Measure Measurements. Existing detectors results are obtained from Sajnani et al. [95].

<table>
<thead>
<tr>
<th>Tool</th>
<th>Type of Clone</th>
<th>Recall(%)</th>
<th>Precision(%)</th>
<th>F-Measure(%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>SourcererCC</td>
<td>Type1/2</td>
<td>99</td>
<td>94.8</td>
<td>94.8</td>
</tr>
<tr>
<td></td>
<td>VST3</td>
<td>93</td>
<td>92</td>
<td>92</td>
</tr>
<tr>
<td></td>
<td>ST3</td>
<td>61</td>
<td>73</td>
<td>73</td>
</tr>
<tr>
<td></td>
<td>MT3</td>
<td>5</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td></td>
<td>WT3/4</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>Type1/2</td>
<td>96</td>
<td></td>
<td>82.5</td>
</tr>
<tr>
<td></td>
<td>VST3</td>
<td>62</td>
<td></td>
<td>67</td>
</tr>
<tr>
<td></td>
<td>ST3</td>
<td>15</td>
<td>≈ 60 – 72 (as reported)</td>
<td>≈ 60 – 72</td>
</tr>
<tr>
<td></td>
<td>MT3</td>
<td>1</td>
<td></td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>WT3/4</td>
<td>0</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>CCFinder</td>
<td>Type1/2</td>
<td>59</td>
<td></td>
<td>72.2</td>
</tr>
<tr>
<td></td>
<td>VST3</td>
<td>62</td>
<td></td>
<td>74</td>
</tr>
<tr>
<td></td>
<td>ST3</td>
<td>31</td>
<td></td>
<td>47</td>
</tr>
<tr>
<td></td>
<td>MT3</td>
<td>12</td>
<td></td>
<td>21</td>
</tr>
<tr>
<td></td>
<td>WT3/4</td>
<td>1</td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>Deckard</td>
<td>Type1/2</td>
<td>91</td>
<td></td>
<td>93 (as reported)</td>
</tr>
<tr>
<td></td>
<td>VST3</td>
<td>82</td>
<td></td>
<td>(Unreported)</td>
</tr>
<tr>
<td></td>
<td>ST3</td>
<td>24</td>
<td></td>
<td>(Unreported)</td>
</tr>
<tr>
<td></td>
<td>MT3</td>
<td>0</td>
<td></td>
<td>(Unreported)</td>
</tr>
<tr>
<td>iClones</td>
<td>Type1/2</td>
<td>91</td>
<td></td>
<td>94</td>
</tr>
<tr>
<td></td>
<td>VST3</td>
<td>82</td>
<td></td>
<td>96</td>
</tr>
<tr>
<td></td>
<td>ST3</td>
<td>24</td>
<td></td>
<td>98</td>
</tr>
<tr>
<td></td>
<td>MT3</td>
<td>0</td>
<td></td>
<td>80</td>
</tr>
<tr>
<td></td>
<td>WT3/4</td>
<td>0</td>
<td></td>
<td>60</td>
</tr>
<tr>
<td>NiCad</td>
<td>Type1/2</td>
<td>100</td>
<td></td>
<td>98</td>
</tr>
<tr>
<td></td>
<td>VST3</td>
<td>100</td>
<td></td>
<td>89 – 98</td>
</tr>
<tr>
<td></td>
<td>ST3</td>
<td>95</td>
<td></td>
<td>87 – 95</td>
</tr>
<tr>
<td></td>
<td>MT3</td>
<td>1</td>
<td></td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>WT3/4</td>
<td>0</td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>Our Method Using Distance Fusion</td>
<td>Type1/2</td>
<td>100</td>
<td></td>
<td>95.1</td>
</tr>
<tr>
<td></td>
<td>VST3</td>
<td>64</td>
<td></td>
<td>76.2</td>
</tr>
<tr>
<td></td>
<td>ST3</td>
<td>94.9</td>
<td></td>
<td>95.3</td>
</tr>
<tr>
<td></td>
<td>MT3</td>
<td>94</td>
<td></td>
<td>92.8</td>
</tr>
<tr>
<td></td>
<td>WT3/4</td>
<td>94.7</td>
<td></td>
<td>95</td>
</tr>
<tr>
<td>Our Method Using Multiplicative</td>
<td>Type1/2</td>
<td>92.1</td>
<td></td>
<td>94.1</td>
</tr>
<tr>
<td></td>
<td>VST3</td>
<td>93.6</td>
<td></td>
<td>92.7</td>
</tr>
<tr>
<td></td>
<td>ST3</td>
<td>95.4</td>
<td></td>
<td>94.7</td>
</tr>
<tr>
<td></td>
<td>MT3</td>
<td>92.8</td>
<td></td>
<td>92.3</td>
</tr>
<tr>
<td></td>
<td>WT3/4</td>
<td>94.3</td>
<td></td>
<td>94.6</td>
</tr>
<tr>
<td>Our Method Using Linear Fusion</td>
<td>Type1/2</td>
<td>92.1</td>
<td></td>
<td>93.9</td>
</tr>
<tr>
<td></td>
<td>VST3</td>
<td>93.2</td>
<td></td>
<td>92.6</td>
</tr>
<tr>
<td></td>
<td>ST3</td>
<td>97.6</td>
<td></td>
<td>96.7</td>
</tr>
<tr>
<td></td>
<td>MT3</td>
<td>95.8</td>
<td></td>
<td>94.7</td>
</tr>
<tr>
<td></td>
<td>WT3/4</td>
<td>95.4</td>
<td></td>
<td>96.2</td>
</tr>
</tbody>
</table>
Appendix B

FEATURES USED IN CHAPTER VI

More AST Features

Table 2.1: More AST features

<table>
<thead>
<tr>
<th>Category</th>
<th>Features</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AST</td>
<td>No. of Annotation Type Member Declarations</td>
<td>Counts &quot;{ Javadoc } { ExtendedModifier } Type Identifier { default Expression }}&quot; in method blocks.</td>
</tr>
<tr>
<td></td>
<td>No. of Anonymous Class Declarations</td>
<td>Counts &quot;{ ClassBodyDeclaration }&quot; in method blocks.</td>
</tr>
<tr>
<td></td>
<td>No. of Annotation TypeDeclaration</td>
<td>Counts &quot;{ Javadoc } { ExtendedModifier } &amp; interface Identifier { AnnotationTypeBodyDeclaration } }&quot; in method blocks.</td>
</tr>
<tr>
<td></td>
<td>No. of String Literals</td>
<td>Counts class StringLiteral extends Expression in method blocks.</td>
</tr>
<tr>
<td></td>
<td>No. of Package Declarations</td>
<td>Counts class PackageDeclaration in method blocks.</td>
</tr>
<tr>
<td></td>
<td>No. of Number Literals</td>
<td>Counts class NumberLiteral extends Expression in method blocks.</td>
</tr>
<tr>
<td></td>
<td>No. of Null Literals</td>
<td>Counts class NullLiteral extends Expression in method blocks.</td>
</tr>
<tr>
<td></td>
<td>No. of Name Qualified Types</td>
<td>Counts &quot;{ Name } { Annotation } SimpleName&quot; in method blocks.</td>
</tr>
<tr>
<td></td>
<td>No. of Modifiers</td>
<td>Counts &quot;public final class Modifier extends ASTNode implements IExtendedModifier&quot; in method blocks.</td>
</tr>
<tr>
<td></td>
<td>No. of Member Refs</td>
<td>Counts &quot;{ Name } &amp; Identifier&quot; in method blocks.</td>
</tr>
<tr>
<td></td>
<td>No. of Marker Annotations</td>
<td>Counts &quot;&amp; TypeName&quot; in method blocks.</td>
</tr>
<tr>
<td></td>
<td>No. of Import Declarations</td>
<td>Counts &quot;{ ImportDeclaration }&quot; in method blocks.</td>
</tr>
<tr>
<td></td>
<td>No. of Javadoc</td>
<td>Counts &quot;{ TagElement }&quot; in method blocks.</td>
</tr>
<tr>
<td></td>
<td>No. of Initializers</td>
<td>Counts &quot;{ static } { Block }&quot; in method blocks.</td>
</tr>
<tr>
<td></td>
<td>No. of Enum Constant Declarations</td>
<td>Counts &quot;{ Javadoc } { ExtendedModifier } Identifier { ( Expression } { AnnotationTypeBodyDeclaration } }&quot; in method blocks.</td>
</tr>
<tr>
<td></td>
<td>No. of Empty Statements</td>
<td>Counts &quot;;&quot; in method blocks.</td>
</tr>
<tr>
<td></td>
<td>No. of Declarations</td>
<td>Counts &quot;{ Annotation }&quot; in method blocks.</td>
</tr>
<tr>
<td></td>
<td>No. of Compilation Units</td>
<td>Counts &quot;{ PackageDeclaration } { ImportDeclaration } { TypeDeclaration } { EnumDeclaration } { AnnotationTypeDeclaration } }&quot; in method blocks.</td>
</tr>
<tr>
<td></td>
<td>No. of Character Literals</td>
<td>Counts &quot;public class CharacterLiteral extends Expression&quot; in method blocks.</td>
</tr>
<tr>
<td></td>
<td>No. of Super Constructor Invocations</td>
<td>Counts &quot;{ Type } { Type } { super } { Expression } }&quot; in method blocks.</td>
</tr>
<tr>
<td></td>
<td>No. of Single Member Annotations</td>
<td>Counts &quot;&amp; TypeName { Expression }&quot; in method blocks.</td>
</tr>
<tr>
<td></td>
<td>No. of Normal Annotations</td>
<td>Counts &quot;&amp; TypeName { MemberValuePair } { MemberValuePair } }&quot; in method blocks.</td>
</tr>
</tbody>
</table>
### BDG Features

**Table 2.2: Some BDG features. “→” means “dependent”**

<table>
<thead>
<tr>
<th>Category</th>
<th>Features</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BDG</td>
<td>No. of ConditionalBranchLoad</td>
<td>Counts ConditionalBranch → Aload in method blocks.</td>
</tr>
<tr>
<td></td>
<td>No. of CondZeroAload</td>
<td>Counts CondZero → Aload in method blocks.</td>
</tr>
<tr>
<td></td>
<td>No. of CondCompare-Aload</td>
<td>Counts CondCompare → Aload in method blocks.</td>
</tr>
<tr>
<td></td>
<td>No. of ConditionalBranchIload</td>
<td>Counts ConditionalBranch → Iload in method blocks.</td>
</tr>
<tr>
<td></td>
<td>No. of CondZeroIload</td>
<td>Counts CondZero → Iload in method blocks.</td>
</tr>
<tr>
<td></td>
<td>No. of CondCompare-Iload</td>
<td>Counts CondCompare → Iload in method blocks.</td>
</tr>
<tr>
<td></td>
<td>No. of ConditionalBranchIstore</td>
<td>Counts ConditionalBranch → Istore in method blocks.</td>
</tr>
<tr>
<td></td>
<td>No. of CondZeroIstore</td>
<td>Counts CondZero → Istore in method blocks.</td>
</tr>
<tr>
<td></td>
<td>No. of CondCompare-Istore</td>
<td>Counts CondCompare → Istore in method blocks.</td>
</tr>
<tr>
<td></td>
<td>No. of ConditionalBranchInc</td>
<td>Counts ConditionalBranch → Inc in method blocks.</td>
</tr>
<tr>
<td></td>
<td>No. of CondZeroInc</td>
<td>Counts CondZero → Inc in method blocks.</td>
</tr>
<tr>
<td></td>
<td>No. of CondCompareInc</td>
<td>Counts CondCompare → Inc in method blocks.</td>
</tr>
<tr>
<td></td>
<td>No. of ConditionalBranchNew</td>
<td>Counts ConditionalBranch → New in method blocks.</td>
</tr>
<tr>
<td></td>
<td>No. of CondZeroNew</td>
<td>Counts CondZero → New in method blocks.</td>
</tr>
<tr>
<td></td>
<td>No. of CondCompareNew</td>
<td>Counts CondCompare → New in method blocks.</td>
</tr>
</tbody>
</table>