Detection and Analysis of Software Clones

By

Abdullah Mohammad Sheneamer

M.S., University of Colorado at 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 at Colorado Springs
in partial fulfillment of the
requirement for the degree of
Doctor of Philosophy
Department of Computer Science

2017

April 12, 2017
This dissertation for Doctor of Philosophy degree by

Abdullah Mohammad Sheneamer

has been approved for the

Department of Computer Science

by

Dr. Jugal Kalita, Computer Science, Chair

Dr. Tim Chamillard, Computer Science

Dr. Albert Glock, Computer Science

Dr. Jonathan Ventura, Computer Science

Dr. Charlie Wang, Electrical and Computer Engineering

Date
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 any machine learning framework using such datasets appear to be ineffective. 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 in order to confound reverse engineering. Obfuscated code detection is challenging due to the availability of code obfuscation tools. We observe a nice resemblance between semantic clones and obfuscated codes also. We apply our clone detection scheme to detect obfuscated codes. We propose a framework that can detect both code clones and obfuscated code as our final contribution. Our results are superior in comparison to state-of-the-art obfuscated code detection methods.
Dedication

To my parents, my brothers, my sisters, my lovely child, and beloved wife.
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 my tough times. I am grateful to all my close 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 for the most of time I was working on this dissertation.
# Table of Contents

1 Introduction .................................................................................................................. 1

1.1 Overview .................................................................................................................... 1

1.1.1 Basic Definitions ................................................................................................. 2

1.2 Types of Clones .......................................................................................................... 3

1.3 Clone Detection Phases ............................................................................................. 4

1.3.1 Code Preprocessing or Lexical Analysis ............................................................... 4

1.3.2 Transformation ...................................................................................................... 5

1.3.3 Match Detection ..................................................................................................... 7

1.3.4 Formatting .............................................................................................................. 7

1.3.5 Filtering ................................................................................................................ 7

1.3.6 Aggregation ........................................................................................................... 7

1.4 Assessment Matrices ................................................................................................. 8

1.4.1 Precision and Recall ............................................................................................ 8

1.4.2 Portability ............................................................................................................. 10

1.4.3 Scalability ............................................................................................................ 10

1.4.4 Comparison Units ............................................................................................... 10

1.4.5 Robustness ........................................................................................................... 11

1.4.6 Language Independence ...................................................................................... 11

1.4.7 Types of Clones .................................................................................................... 11

1.5 Application of clone detection .................................................................................... 12

1.5.1 Clone Avoidance .................................................................................................. 12
4.3.1 AST and PDG: Novel features for clone detection .......................... 95
4.3.2 Fusion of Block Features .................................................. 100
4.3.3 Clone Detection Framework ............................................... 102
4.4 Experimental Evaluation ................................................... 102
4.4.1 Datasets ........................................................................ 103
4.4.2 Classification Models ...................................................... 104
4.4.3 Evaluation ...................................................................... 106
4.4.4 Performance of different classifiers ................................. 106
4.5 Chapter Summary ............................................................... 112

5 Automated Labeling Type-IV Code Clone Using Java ByteCode 115
5.1 Introduction ........................................................................ 115
5.2 Prior Research ..................................................................... 116
5.3 Java Bytecode Overview .................................................... 117
5.4 The Proposed Method ......................................................... 117
5.5 Chapter Summary ............................................................... 122

6 Obfuscated Code Detection- An Application of Semantic Clone Detection Scheme123
6.1 Introduction ........................................................................ 123
6.2 Background ....................................................................... 125
6.3 Prior Research .................................................................... 128
6.4 An Integrated Detection Framework ................................. 129
6.4.1 Java ByteCode : Low Level Features ............................. 131
6.4.2 Source Code Features .................................................... 134
6.4.3 Fusion of Code Features .................................................. 137
6.4.4 Code Similarity as a Feature ............................................. 138
6.4.5 A New Code Obfuscation and Clone Detection Scheme .......... 140
6.5 Experimental Evaluation ................................................... 141
  6.5.1 Datasets ................................................................. 142
  6.5.2 Ensemble Classification Model ......................................... 144
  6.5.3 Experimental Results .................................................. 145
  6.5.4 Performance comparison .............................................. 149
6.6 Chapter Summary ........................................................... 153

7 Conclusion and Future Work .................................................. 156
  7.1 Conclusion ................................................................... 156
  7.2 Future Work .................................................................. 157

References ............................................................................. 160

Appendix A Features and Results of Chapter 4 ............................. 176
  A.1 Traditional Features ......................................................... 176
  A.2 AST Features ................................................................. 177
  A.3 PDG Features ................................................................. 178

Appendix B Features of Chapter 6 .............................................. 178
  A.4 Results of combinations of pair instances vectors .................. 179
  A.5 Results of BigCloneBench Recall, Precision and F-Measure Measurements 180
  B.1 More AST Features ......................................................... 181
B.2 BDG Features ........................................... 182
List of Tables

2.1 Techniques for clone detection ........................................... 22
2.2 Summary of textual approaches ........................................... 23
2.3 Summary of lexical approaches ......................................... 31
2.4 Summary of syntactical approaches ................................. 41
2.5 Summary of semantic approaches ................................... 53
2.6 Evaluations of clone detection tools .......................... 60

3.1 Target software systems .................................................. 70
3.2 Our approaches for Type-III clones results .................... 80
3.3 Our approaches for all Type-I, II and III clones results .... 80

4.1 Different classification techniques used ....................... 105

5.1 Brief description of our Java code clone corpus ................. 121

6.1 Byte Code Conditional Statements .................................. 132
6.2 Categorization of Byte Code instructions .................... 133
6.3 Brief description of Java obfuscated datasets ................. 144

A.1 Traditional features ...................................................... 176
A.2 AST features ................................................................. 177
A.3 PDG features. ”→” means ”dependent” ............................... 178

A.4 Results of distance and multiplicative combinations of pair instances vectors 179
A.5 BigCloneBench Recall, Precision and F-Measure Measurements. ...... 180
B.1 More AST features ......................................................... 181

B.2 Some of BDG features. ”→” means ”dependent” ................. 182
List of Figures

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

2.1 CP-Miner Process Steps ............................................. 35
2.2 Example of coarse-grained clone detection ......................... 46

3.1 The proposed method ................................................ 64
3.2 The results of Type-I and Type-II ................................ 74
3.3 The results of Type-III .............................................. 75
3.4 The results of Type-I,II, and III ................................ 76
3.5 Cosine similarity time ................................................ 79
3.6 Levenshtein distance time .......................................... 79
3.7 Levenshtein distance threshold ................................... 82
3.8 Cosine threshold ...................................................... 83

4.1 Example of AST derived from code block ......................... 98
4.2 Program dependency graph ......................................... 98
4.3 Share of different categories of features used .................. 100
4.4 Workflow of the proposed clone detection framework ............ 103
4.5 Performance all the candidate classifiers with different feature fusions . 107
4.6 Performance of three best classifiers with synthetic and semantic features . 108
4.7 Learning Curve ....................................................... 110
4.8 Performance of Random Forest and Rotaion Forest Feature Selections .... 111
4.9 Performance comparison of different detection methods . . . . . . . . . . 112

5.1 A new framework for labelling semantic code clones . . . . . . . . . . . 119

6.1 A BDG showing control and data dependency among the instructions. . . 133

6.2 Share of categories of features used. . . . . . . . . . . . . . . . . . . . . 139

6.3 Workflow of the proposed dual detection framework . . . . . . . . . . . 141

6.4 Effectiveness of the framework on detecting clone code using a model that
was built by training set and another test set. . . . . . . . . . . . . . . . . . 142

6.5 Java obfuscated dataset generation steps . . . . . . . . . . . . . . . . . . 144

6.6 Effectiveness of the framework on detecting obfuscated code using a model
that was built by training set and another test set. . . . . . . . . . . . . . . . . 144

6.7 Performance of detection framework on clone datasets with different fea-
tures combinations . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 146

6.8 Effectiveness of the framework on detecting obfuscated code using features
fusions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 147

6.9 Learning Curve: Performance of ensemble classifier on clone dataset with
varying number of features. . . . . . . . . . . . . . . . . . . . . . . . . . . . . 147

6.10 Performance of ensemble approach on clone dataset after feature selection
using *MDI* . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 148

6.11 Effectiveness of ensemble classifier on detecting obfuscated codes after
feature selection using *MDI*. . . . . . . . . . . . . . . . . . . . . . . . . . . . 149

6.12 Prediction effectiveness of proposed framework in comparison to state-of-
the-art clone detectors in terms of F-Score . . . . . . . . . . . . . . . . . . 151
6.13 Effectiveness of various obfuscated code detection tools on ObsCode dataset

6.14 Tool performance comparison on the PacMan data in terms of accuracy, which original program compared to obfuscated programs for each clone detection technique. The obfuscation are abbreviated as follows: C–contraction, E–expansion, L–loop transformation, R–renaming [98] . . . . . . . . . . 154
Chapter 1

INTRODUCTION

1.1 Overview

If two fragments of source code are identical or similar to each other, they are called code clones. 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 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 [91].

Code clones are used frequently because they can be created fast, and easily inserted with little expense [114]. 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 of code segment. All these make it a difficult job to maintain a large system [75, 87]. Ad-
ditional reasons that make code clone detection essential are the following. 1) Detecting
cloned code may help detect malicious software [106]. 2) Code clone detection may find
similar code and help detect plagiarism and copyright infringement [50, 75, 106]. 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 pro-
gram that impact other issues that arise when code is duplicated all over the system [14].

1.1.1 Basic Definitions

Each paper in the literature defines clones in its own way [87]. 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 statements that are considered interesting,
but it may contain fewer than five statements. It can contain a function or a method, `begin-
end` blocks or a sequence of statements.

**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.
1.2 Types of Clones

There are two groups of clones. The first group refers to two code fragments which are similar based on their text [10,54]. There are three types within the first group as shown in 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 1.1.

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

### 1.3 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

#### 1.3.1 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.

1. **Remove uninteresting pieces of code.** All elements in the source code that have no bearing in 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.

### 1.3.2 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, 50, 67]. 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, 105].

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 [58].

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. [50], which has transformation rules for C++ remove template parameters. The rule is \(\text{Name '<ParameterList>'} \rightarrow \text{Name}\). For example, \(\text{foo<int>}\) is transformed into \(\text{foo}\) [50].

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.
1.3.3 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 [50].

1.3.4 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.

1.3.5 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 [95]. The false positives can also be filtered out by automated heuristics based on length, diversity or frequency.

1.3.6 Aggregation

This phase is optional. It can be done in 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 \((C_1, C_2, C_3)\).

### 1.4 Assessment Matrices

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 comparing evaluations of tools and summary of evaluation approaches respectively. Some evaluation metrics are discussed below.

#### 1.4.1 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:

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

(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%.

### 1.4.1.1 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.

### 1.4.1.2 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.

1.4.2 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.

1.4.3 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. Metrics-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.

1.4.4 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.

1.4.5 Robustness

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

1.4.6 Language Independence

Language independence 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.

1.4.7 Types of Clones

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

1.5.1 Clone Avoidance

Two approaches are discussed to deal with 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. [62] use a code clone detection tool 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.

1.5.2 Plagiarism Detection

Code clone detection approaches can be used in plagiarism detection of software code. Dup [6] is a technique that is used for finding near matches of long sections of software code. JPlag [82] 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. [116] propose a count-based clone detection technique called CMCD. CMCD has been used to detect plagiarisms in homeworks of students.
1.5.3 Code Obfuscation Detection

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

Semantics of a program written by anonymous programmer is difficult to identify obfuscated programs. A number of methods and software tools are available for obfuscated code. In chapter 6, we proposed a novel integrated framework for detecting both java code clones and java obfuscated codes. We captured the semantics of program codes using low and high level program features derived from Byte Code, the data and conditional dependency with in the byte code, AST and PDG.

1.5.4 Bug Detection

Code clone detection techniques can also help in bug detection. CP-Miner [67] has been used to detect bugs. Higo et al. [40] propose an approach to efficiently detect bugs that are caused by copy-paste programming.

1.5.5 Code Compacting

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

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

1.5.7  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 an automatic clone detection using large models in graph theory.

1.6  Motivation and Main Contributions

In the software engineering life cycle, code duplicating is not only increased maintenance costs but also considered defect-prone 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 the following, four main works are conducted to solve 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. Also, we build a syntactic and semantic model for a clone detector. The model uses features, which are extracted from Abstract Syntax Tree (AST), Program Dependency Graph (PDG), Bytecode files (BC), and bytecode Dependency Graph (BDG). We elaborate these works as follows:

(a) **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 tailor the Levenshtein distance algorithm to code clone detection. Levenshtein distance is a string 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.

(b) **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.

(c) **Automated Labeling Type-IV Code Clone Using Java ByteCode.**

• Majority of the clone datasets used for machine learning 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.*

(d) **Obfuscated Code Detection- An Application of Semantic Clone Detection Scheme.**

• We propose an integrated framework for detecting java code clone and obfuscated codes using program or code features extracted from target pair of codes to be matched for possible detection of clones or obfuscation.

• We use high level source code features from Abstract Syntax 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.

1.7 Thesis Outline

The remaining of the thesis is organized as follows:

In Chapter 2, we present literature review, which covers the background and related work on code clone detection, introducing the state-of-the-art of code clone detectors and identifying the detection mechanisms.

Chapter 3 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 4, 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 5, we introduce a new dataset of clone references, which is a set of correct clones for Type-IV clones.

In Chapter 6, we propose a dual detection framework for detecting both Java code obfuscation and clone using an integrated machine learning scheme. BDG is alternative representation of semantics or meaning of a Java program. So, the novel
Java byteCode dependency graph (BDG) features are extracted to detect both Java code obfuscation and clone.

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 potential can be developing a new hybrid technique. The use of evaluation of clone detection techniques based on recall, precision and F-measure metrics, scalability, portability and clone relation in order to choose the right technique for a specific task and several evaluation metrics can be used.

In Chapter 7, we conclude the works in our thesis, and propose the potential directions in the future.

1.8 Publication List

All publications and submitted papers are listed as follows.

(a) Published Papers.


(b) **Submitted Papers.**


2.1 Introduction

Recent research [6, 50, 67, 117] with large software systems [67] has detected that 22.3% of Linux code has clones. Kamiya has et al. [50] 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. [72] have also reported that 5% - 20% code is cloned. Code clone detection can be useful for code simplification, code maintainability, plagiarism detection [69, 82], copyright infringement detection, malicious software detection and detection of bug reports. Many code clone detection techniques have been proposed [87]. The focus of this chapter is to present a review of such clone detection techniques.
2.2 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.

2.2.1 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 [28], Ducasse et al. [27], Koschke et al. [57], NICAD by Roy and James [93] and SSD by Seunghak and Jeong [65].
<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></td>
<td>1995</td>
<td>[6]</td>
</tr>
<tr>
<td></td>
<td></td>
<td><em>Duploc</em></td>
<td>1999</td>
<td>[28]</td>
</tr>
<tr>
<td></td>
<td></td>
<td><em>NICAD</em></td>
<td>2008</td>
<td>[93]</td>
</tr>
<tr>
<td></td>
<td></td>
<td><em>SDD</em></td>
<td>2005</td>
<td>[65]</td>
</tr>
<tr>
<td>Lexical</td>
<td>Token</td>
<td><em>CCFinder</em></td>
<td>2002</td>
<td>[50]</td>
</tr>
<tr>
<td></td>
<td></td>
<td><em>CP-Miner</em></td>
<td>2006</td>
<td>[67]</td>
</tr>
<tr>
<td></td>
<td></td>
<td><em>Boreas</em></td>
<td>2012</td>
<td>[117]</td>
</tr>
<tr>
<td></td>
<td></td>
<td><em>FRIISC</em></td>
<td>2012</td>
<td>[78]</td>
</tr>
<tr>
<td></td>
<td></td>
<td><em>CDSW</em></td>
<td>2013</td>
<td>[77]</td>
</tr>
<tr>
<td>Syntactic</td>
<td>Tree</td>
<td><em>CloneDr</em></td>
<td>1998</td>
<td>[10]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Wahler <em>et al.</em></td>
<td>2004</td>
<td>[105]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Koschke <em>et al.</em></td>
<td>2006</td>
<td>[57]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Jiang <em>et al.</em></td>
<td>2007</td>
<td>[47]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Hotta <em>et al.</em></td>
<td>2014</td>
<td>[44]</td>
</tr>
<tr>
<td>Metric</td>
<td>Tree</td>
<td>Mayrand <em>et al.</em></td>
<td>1996</td>
<td>[72]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Kontogiannis <em>et al.</em></td>
<td>1996</td>
<td>[56]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Kodhai, <em>et al.</em></td>
<td>2010</td>
<td>[53]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Abdul-El-Hafiz <em>et al.</em></td>
<td>2012</td>
<td>[1]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Kanika <em>et al.</em></td>
<td>2013</td>
<td>[86]</td>
</tr>
<tr>
<td>Semantic</td>
<td>Graph</td>
<td><em>Duplix</em></td>
<td>2001</td>
<td>[58]</td>
</tr>
<tr>
<td></td>
<td></td>
<td><em>GPLAG</em></td>
<td>2006</td>
<td>[69]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Higo and Kusumoto</td>
<td>2009</td>
<td>[41]</td>
</tr>
<tr>
<td>Hybrid</td>
<td></td>
<td>ConQAT</td>
<td>2011</td>
<td>[45]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Agrawal <em>et al.</em></td>
<td>2013</td>
<td>[2]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Funaro <em>et al.</em></td>
<td>2010</td>
<td>[34]</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 of Clones</th>
<th>Types of Independence</th>
<th>Language Independent</th>
<th>Output Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dup</td>
<td>[6] 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 of lines</td>
<td>Type-I Type-II</td>
<td>Needs lexer</td>
<td>Text</td>
</tr>
<tr>
<td>Duploc</td>
<td>[28] Removes comments and all white space</td>
<td>Sequence of lines of Dynamic Pattern Matching</td>
<td>Dynamic Pattern Matching</td>
<td>$O(\text{line})$ where $n$ is number of input lines</td>
<td>Line</td>
<td>Type-I Type-II</td>
<td>Needs lexer</td>
<td>Text</td>
</tr>
<tr>
<td>NICAD</td>
<td>[93] Pretty-printing 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</td>
<td>Text</td>
<td>Type-I Type-II Type-III</td>
<td>Needs parser</td>
<td>Text</td>
</tr>
<tr>
<td>SDD</td>
<td>[65] No transformation inverted index and index N-neighbor distance</td>
<td>No transformation inverted index and index N-neighbor distance</td>
<td>No transformation inverted index and index N-neighbor distance</td>
<td>$O(n)$</td>
<td>Chunks of source code</td>
<td>Type-I Type-II Type-III</td>
<td>No lexer/parser needs</td>
<td>Visualization similar of code</td>
</tr>
</tbody>
</table>

2.2.1.1 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 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 introduces 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 re-
naming function. Parameterized matches can be detected using p-strings, which are strings that contain ordinary characters from an alphabet \( \Sigma \), and parameter characters from a finite alphabet \( \Pi \). Dup implements a p-match algorithm. The lexical analyzer produces a string containing non-parameter symbol 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 parameter 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 [9,95], to represent parameter strings in a p-suffix tree. In Definition 4, the \( f \) function, called transform, computes the \( j \)-th symbol value of \( p\text{-suffix}(S,i) \) 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 a p-string \( \hat{P} \) are p-match of each other if and only if \( \text{prev}(P) = \text{prev}(\hat{P}) \), where \( \text{prev} \) 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 \( \hat{P} \) 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 \( p\text{-suffix}( \hat{P}, i) \).

**Definition 4.** If \( b \) belongs to alphabet \( \Sigma \) union alphabet \( \Pi \), \( f(b,j)=0 \) 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].

2.2.1.2 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 se-
quences ‘abcghdجي’ and ‘abcfklنجي’ are compared, the gap is of length 4 because the lengths of the two non-duplicated subsequences ghдж and fklن are 4. False positives are averted 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 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. [57] 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 [57] because detecting bugs requires semantic information and Duploc detects just syntactic clones.

2.2.1.3 NICAD by Roy and James.

Roy and James [93] develop a text-based code clone detection technique called Accurate Detection of Near-miss Intentional Clones (NICAD). The NICAD tool [93] 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) * PUST / 100 and size (clone 1) + size (clone 2) * PUST / 100. NICAD can detect exact and near-miss clones at the block level of granularity. NICAD has high precision and recall [94]. It can detect even some exact function clones that are not detected by the exact matching function used by a tree-based technique [93]. 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.

2.2.1.4 SDD by Seunghak and Jeong.

Seunghak and Jeong [65] 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) [46]. 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:

$$\text{Nearest Neighbor Distance} = \frac{\sum_{i} \left[\text{Min}(d_{ij})\right]}{N}$$

(2.2)

where $N$ is the number of points and $\text{Min}(d_{ij})$ 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.
2.2.1.5 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* [93] 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* [65] 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, 67]. 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 [50].
2.2.2 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 [50] by Kamiya et al., CP-Miner [67, 93] by Zhenmin et al., Boreas [117] by Yong and Yao, FRISC [117] by Murakami et al., and CDSW [117] 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.

2.2.2.1 CCFinder by Kamiya et al.

Kamiya et al. [50] 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.
<table>
<thead>
<tr>
<th>Tool</th>
<th>Transformation Method</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><strong>CCFinder</strong> [80]</td>
<td>Move 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><strong>CP-Miner</strong> [67]</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><strong>Boreas</strong> [117]</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>Clustering</td>
</tr>
<tr>
<td><strong>FRISC</strong> [78]</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><strong>CDSW</strong> [77]</td>
<td>Remove whitespace, comments; map from transformed sequence into original, and parameter replace</td>
<td>Hash values for every statement</td>
<td>Smith-Waterman Alignment</td>
<td>(O(nm)) where (n) is length of first token sequences and (m) is length of second token sequences</td>
<td>Token Sequences</td>
<td>Type-I Type-II Type-III</td>
<td>Needs lexer</td>
<td>Clone pairs</td>
</tr>
</tbody>
</table>
CCFinder applies several metrics to detect interesting clones. These metrics are given below. 1) The length of a code fragment that can be used by the number of tokens or the number of lines of the code fragment. 2) Population size of a clone class: It is 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. It 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 [52]. 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, the layout of the code is changed. CCFinder performs a more suitable transformation than the line-by-line method [50]. 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, 50, 67]. CCFinder uses a suffix tree algorithm, and so it cannot handle statement insertions and deletions in code clones [67].

2.2.2.2 CP-Miner by Li et al.

Li et al. [67] 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 [113]. *CP-Miner* implements two functions.

(i) **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 [113] 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.

\[
UnRenamed\_IDRate = \frac{\text{NumUnchanged\_ID}}{\text{TotalUnchanged\_ID}} \tag{2.3}
\]

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 all occurrences of identifiers have been changed, and if *UnRenamed\_IDRate* is 1, it means all occurrences of the identifier remain unchanged. *CP-Miner* can only detect forgot-to-change bugs. This means if the programmer has forgotten to modify or insert some extraneous statements to the new
copy-pasted segment, \textit{CP-Miner} would not detect the bug because the changed code fragments are now too different from the others [67]. This approach can detect similar sequences of tokenized statements and avert redundant comparisons, and as a result, it detects code clones efficiently, even in millions of code lines.

\textit{CP-Miner} overcomes some limitations of \textit{CCFinder} and detects more copy-pasted segments than \textit{CCFinder} does. However, \textit{CCFinder} does not detect code clones that are related to bugs as \textit{CP-Miner} does because \textit{CP-Miner} uses an unchanged ratio threshold. \textit{CCFinder} does not completely filter false positives and it detects many tiny cloned code blocks which seem to be predominantly false positives. Because \textit{CP-Miner} handles statement insertions and modifications, \textit{CP-Miner} can detect 17-52\% more code clones than \textit{CCFinder}. Unfortunately, the frequent subsequence mining algorithm that \textit{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 [67].

2.2.2.3 \textit{Boreas} by Yang and Yao.

Yang and Yao [117] use a token-based approach called \textit{Boreas} to detect clones. \textit{Boreas} uses a novel counting method to obtain characteristic matrices that identify program segments effectively. \textit{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 the variable in the \( i \)-th CE. CV is also called a characteristic vector. 3) Counting Matrix (CM), which contains all \( n \) (n-variables) CVs in code fragments and is an \( n \times m \) Count Matrix. Boreas uses the cosine of the two vectors angle to
compare similarity:

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

(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. \( PropSimilarity \) is proportional similarity between \( C_1 \) and \( C_2 \), which are two occurrences counts:

\[ PropSimilarity = \frac{1}{(C_1 + 1)} + \frac{C_2}{(C_1 + 1)}. \]  

(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.

2.2.2.4 \textit{FRISC} by Murakami et al.

Murakami et al. [78] 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. Then, 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 is higher than the precision without by 29.8%, but the recall decreases by 2.9%. FRISC has higher recall and precision than CCFinder but the precision is lower than CloneDr [10] and CLAN [73].

2.2.2.5 CDSW by Murakami et al.

Murakami et al. [77] develop another token-based technique, which detects Type-III clones (gapped clones) using the Smith-Waterman algorithm [101], called CDSW. 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 Longest Common Subsequences (LCS) 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 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. 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.

2.2.2.6 SourcererCC by Sajnani et al.

SourcererCC [96] 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.
2.2.2.7 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 [78]. 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.
2.2.3 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 al., Wahler [105], Koschke [57], Jiang [47], Mayrand et al. [72], Kontogiannis et al. [56], Kodhai, et al. [53], Abdul-El-Hafiz [1] and Kanika et al. [86].

2.2.3.1 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.

2.2.3.1.1 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 generalizing other clone combinations. The method splits sub-trees using
<table>
<thead>
<tr>
<th>Author/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>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>
<td>Clone pairs</td>
</tr>
<tr>
<td>Wahler [105]</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>
<td>AST nodes</td>
</tr>
<tr>
<td>Koschke [57]</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>
<td>Text</td>
</tr>
<tr>
<td>Jiang et al. [47]</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>$O(</td>
<td>T_1</td>
<td></td>
<td>T_2</td>
<td>[d_1d_2])$, where $</td>
</tr>
<tr>
<td>Hotta et al. [44]</td>
<td>Parse source code to extract blocks using JDT</td>
<td>Hashed blocks</td>
<td>Group blocks based on hash values</td>
<td>N/A</td>
<td>Blocks</td>
<td>Type-I, Type-II</td>
<td>Needs parser</td>
<td>Clone pairs, Clone classes</td>
</tr>
<tr>
<td>Mayrand et al. [72]</td>
<td>Parse to AST then (IRL)</td>
<td>Metrics</td>
<td>21 function metrics</td>
<td>Polynomial complexity</td>
<td>Metrics for each function</td>
<td>Type-I, Type-II, Type-III</td>
<td>Needs Dartix tool</td>
<td>Clone pairs, Clone classes</td>
</tr>
<tr>
<td>Kontogiannis et al. [56]</td>
<td>Transform to feature vectors</td>
<td>Feature vectors</td>
<td>Use numerical comparisons of metric values and dynamic programming (DP) using minimum edit distance</td>
<td>$O(n^2)$ for Naïve approach and $O(nm)$ for DP-model</td>
<td>Metrics of a begin-end block</td>
<td>Type-I, Type-II, Type-III</td>
<td>Needs a parser and an additional tool</td>
<td>Clone pairs</td>
</tr>
<tr>
<td>Kodhai, et al. [53]</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 a parser</td>
<td>Clone pairs and clusters</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>A language independent</td>
<td>Clone Classes</td>
</tr>
<tr>
<td>Kanika et al. [86]</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>
<td>Output mapped into Excel sheets</td>
</tr>
</tbody>
</table>
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:

\[ \text{Similarity} = \frac{2SN}{(2SN + \text{LEFT} + \text{RIGHT})} \]  \hspace{1cm} (2.6)

where \( SN \) is the number of shared nodes, \( \text{LEFT} \) is the number of nodes in sub-tree1 and \( \text{RIGHT} \) 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.

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.

2.2.3.1.2 Wahler et al.

Wahler et al. [105] 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 transaction, each of which has a number of features
called 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:

\[
\sigma(I) = \frac{|\{T \in D \mid I \subseteq T\}|}{|D|} \geq \sigma
\]  

(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 \( I_1 = b_1...b_k \), \( I_2 = b_2...b_{k-1} \).

### 2.2.3.1.3 Koschke et al.

Koschke et al. [57] also detect code clones using an abstract syntax tree (AST). Their method finds syntactic clones by pre-order traversal, applies suffix tree de-
tection 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 detection 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 [57] detects clones in less time.

2.2.3.1.4 Deckard by Jiang et al.

Jiang et al. [47] 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 structure 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 [47] is language independent with lower speed than the Boreas tool, discussed in Subsection 2.2.2, because of less set-up time and less comparison time [67, 117]. Deckard also requires constructing ASTs, which requires more time.
2.2.3.1.5 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 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 of `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.

2.2.3.2 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 [53, 72, 73].
Figure 2.2: Example of coarse-grained clone detection. [44]

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 [53]. Pairs of similar methods are also detected by comparison of the metric values, which are stored in the same database.
2.2.3.2.1 Mayrand et al.

Mayrand et al. [72] 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 codes 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. [80] extend Mayrand’s tool to find Java clones using a similar metric-based algorithm.

2.2.3.2.2 Kontogiannis et al.

Kontogiannis et al. [55] propose a way to measure similarity between two pieces of source code using an abstract patterns 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 [56]. 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.
2.2.3.2.3 Kodhai et al.

Kodhai et al. [53] 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.

2.2.3.2.4 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) Pre-processing 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.

2.2.3.2.5 MCD Finder by Kanika et al.

Kanika et al. [86] use a metric-based approach to develop the MCD Finder 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 [86] 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.

2.2.3.2.6 Wang et al.

Wang et al. [107] propose a syntactic clone detection method using a Bayesian Network framework with a set of features such as history, code and destination fea-
Yang et al. [115] proposed a user feedback based learning model, FICA to detect Type-I, II, and III clones. They 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 [110]. 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.

3) The measurement phase performs a comparison on the similarity of the metric values.

### 2.2.3.3 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 the 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 clones code. Kodhai et al.’s approach only detects Type-I and Type-II clones. 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 byte code 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.

### 2.2.4 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 Hor-
witz [54], Duplix [58] by Krinke, GPLAG [69] by Liu et al., Higo and Kusumoto [41], Hummel et al. [45], Funaro et al. [34], and Agrawal et al. [2].

2.2.4.1 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 1.3.2. 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 [99]. Clones can be detected as isomorphic subgraphs [54]. 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 [69].

**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 [69].
<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 Howitz [54]</td>
<td>PDGs using CodeSurfer</td>
<td>PDGs</td>
<td>Isomorphic PDG sub-graph matching using backward slicing</td>
<td>N/A</td>
<td>PDG node</td>
<td>Type-III Type-IV</td>
<td>Needs tool for converting source code to PDGs</td>
</tr>
<tr>
<td>Duplix [58]</td>
<td>To PDGs</td>
<td>PDGs</td>
<td>K-length patch algorithm</td>
<td>Non-polynomial complexity</td>
<td>PDG Subgraphs</td>
<td>Type-I Type-IV</td>
<td>Needs tool for converting source code to PDGs</td>
</tr>
<tr>
<td>GPLAG [69]</td>
<td>PDGs using CodeSurfer</td>
<td>PDGs</td>
<td>Isomorphic PDG sub-graph matching algorithm</td>
<td>NP-Complete</td>
<td>PDG Node</td>
<td>Type-I Type-II Type-III</td>
<td>Needs tool for converting source code to PDGs</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</td>
<td>Type-III</td>
<td>Needs tool for converting source code to PDGs</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</td>
<td>String clones</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</td>
<td>Text clones</td>
</tr>
</tbody>
</table>
2.2.4.1.1 Komondoor and Horwitz

Komondoor and Horwitz [54] use program slicing [109] 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 [54]. 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.

2.2.4.1.2 Duplix by Krinke

Krinke [58] finds maximal similar PDG subgraphs with high precision and recall. Krinke’s approach is similar to the Komondoor and Horwitz approach [54] although [54] starts from every pair of matching nodes and uses sub-graphs that are not maximal and are just subtrees unlike the ones in [58]. 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.
2.2.4.1.3 **GPLAG by Liu et al.**

Liu *et al.* [69] 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* [82] and are unable to detect disguised plagiarized code well. Plagiarism disguises may include the following [69]. 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. In addition, an `for (int i=0; i<10; i++) {a=b-c;}` block can be replaced by `while (i<10) {a=b-c; i++; }`. 5) Code Insertion, where additional code may be inserted to disguise plagiarism without affecting the results.

2.2.4.1.4 **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.

2.2.4.2 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.

2.2.4.2.1 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 [89]. 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.
2.2.4.2.2 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.

2.2.4.2.3 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 [117], 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.
2.2.4.3 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 Komondoor and Horwitz tool is that it can detect gapped clones. Komondoor and Horwitz 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 syntactical 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.

2.3 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 new 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 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 stare 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.
Table 2.6: Evaluations of clone detection 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>Support Language</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>Depends how comparison is done</td>
<td>Low</td>
<td>Tree/Suffix-tree based token matching</td>
<td>Needs a parser</td>
<td>C, C++, Java, COBOL</td>
<td>Low</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>Low</td>
<td>Tree/Suffix-tree based token matching</td>
<td>Language independent</td>
<td>Java, C, C++</td>
<td>N/A</td>
<td>N/A</td>
<td>Type-I, II</td>
<td>Clone Detection</td>
</tr>
<tr>
<td>CCFinder</td>
<td>High</td>
<td>Low</td>
<td>Medium</td>
<td>High</td>
<td>Medium</td>
<td>Token/Suffix-tree based token matching</td>
<td>Needs lexer and trans. rules</td>
<td>C, C++, Java, COBOL</td>
<td>Medium Medium</td>
<td>Medium</td>
<td>Type-I, II, and week 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 matching</td>
<td>Needs lexer</td>
<td>C, C++, Java</td>
<td>Medium Medium</td>
<td>Medium</td>
<td>Type-I, II</td>
<td>Clone detection</td>
</tr>
<tr>
<td>Duploc</td>
<td>High</td>
<td>Low</td>
<td>High</td>
<td>Depends how comparison algos.</td>
<td>Low</td>
<td>Line/Pattern Matching</td>
<td>Needs lexer</td>
<td>Independent</td>
<td>N/A</td>
<td>N/A</td>
<td>Type-I, and weak Type-II</td>
<td>Clone detection</td>
</tr>
<tr>
<td>NCAD</td>
<td>High</td>
<td>High</td>
<td>High</td>
<td>Depends on comparison alg.</td>
<td>High</td>
<td>Line/Longest Common Subsequence (LCS)</td>
<td>Needs parser</td>
<td>Java</td>
<td>Medium Medium</td>
<td>Type-I,III</td>
<td>Clone detection</td>
<td></td>
</tr>
<tr>
<td>DECKARD</td>
<td>Low</td>
<td>High</td>
<td>Low</td>
<td>Depends how comparison is done</td>
<td>Low</td>
<td>Hybrid/Tree-matching</td>
<td>Context-free grammar</td>
<td>C, Java</td>
<td>Medium Low</td>
<td>Type-I,III</td>
<td>Clone Detection</td>
<td></td>
</tr>
<tr>
<td>CLAN</td>
<td>Medium</td>
<td>Medium</td>
<td>Low</td>
<td>High</td>
<td>Medium</td>
<td>Metrics/21 function metrics</td>
<td>Need a tool (DATRIX)</td>
<td>C</td>
<td>High</td>
<td>High</td>
<td>Type-I, Weak Type-I,III</td>
<td>Clone and plagiarism detection</td>
</tr>
<tr>
<td>iClones</td>
<td>Low</td>
<td>Medium</td>
<td>High</td>
<td>High</td>
<td>Low</td>
<td>Tokens</td>
<td>Needs parser</td>
<td>Java</td>
<td>Medium Low</td>
<td>Type-I, Weak Type-I,III</td>
<td>Clone Detection</td>
<td></td>
</tr>
<tr>
<td>CDSW</td>
<td>High</td>
<td>Low</td>
<td>Medium</td>
<td>High</td>
<td>Low</td>
<td>Tokens/Smith-waterman algorithm</td>
<td>Needs lexer</td>
<td>Java</td>
<td>High</td>
<td>High</td>
<td>Type-I,II</td>
<td>Clone detection</td>
</tr>
<tr>
<td>Course-grained</td>
<td>Low</td>
<td>Low</td>
<td>Medium</td>
<td>Medium</td>
<td>Low</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>SorcererCC</td>
<td>High</td>
<td>Low</td>
<td>High</td>
<td>High</td>
<td>High</td>
<td>Token comparison</td>
<td>Needs parser</td>
<td>Java</td>
<td>High</td>
<td>High</td>
<td>Type-I,II, Very weak Type-IV</td>
<td>Clone detection</td>
</tr>
<tr>
<td>Our Cont. Approach1</td>
<td>Medium</td>
<td>High</td>
<td>Low</td>
<td>Medium</td>
<td>Medium</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 Cont. 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 Medium</td>
<td>Type-I,II,IV</td>
<td>Clone and plagiarism detection</td>
<td></td>
</tr>
<tr>
<td>Our Cont. 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,IV</td>
<td>Clone and plagiarism detection</td>
</tr>
</tbody>
</table>


Acknowledgments

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 137.10(2016).
Chapter 3

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

3.1 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 fine-grained. 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 [77].

We implement the proposed method and evaluate it by using Murakami’s benchmark dataset [77]. 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 Section 3.2, the proposed method is discussed in detail. The similarity measures are described in Section 3.3. Experiment design is discussed in Section 3.4. We discuss the experiments we perform in Section 3.4. Discussions on our approach are covered in Subsection 3.6. Finally, the chapter is concluded in Section 3.7.

3.2 The Proposed Method

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

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 LevDist

```plaintext
1: procedure LevenshteinDistance(B_1, B_2)
2:   define LD[n+1][m+1]
3:   set LD[i][0..m] ← i
4:   set LD[0..n][j] ← j
5:   for j ← 0, n do
6:     b_1 ← B_1.getLine(i−1)
7:     for j ← 0, m do
8:       b_2 ← B_2.getLine(j−1)
9:       if b_1 = b_2 then
10:          score ← 0
11:       else
12:          score ← 1
13:     end if
14:     LD[i][j] ← Min(LD[i−1][j]+1, LD[i][j−1]+1, LD[i−1][j−1] +cost)
15:   end for
16: end procedure
17: return Sim ← 1 − \frac{LD(b_1,b_2)}{Max(len(b_1,b_2))} × 100
```

To explain our steps, we use the two program fragments given in Figure 3.1(a) as running example.

### 3.2.1 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
Algorithm 2 Type-III Clone Detection

1: procedure CloneDetection(BlockFragments)
2:   Clones ← 0
3:   for i ← 0, BlockFragments.Length do
4:     for j ← 0, BlockFragments.Length do
5:       Cosine(B_i, B_j)
6:       if Sim ≥ 70% then
7:         Clones ← Clones + 1
8:       end if
9:     end for
10:   end for
11: return Clones
12: end procedure

the original files and 3.1(b) gives the two program fragments after lexical analysis and normalization. Identifiers have been replaced by the $ sign.

3.2.2 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.

3.2.3 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.

\(^1\)http://www.eclipse.org/jdt/
or cosine similarity. These two similarity measures are discussed in Section 3.3. In 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 a simply 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.

3.2.4 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 Section 3.3.

3.2.5 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.
3.2.6 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 \times freq1
24:   VectorBlock2 ← VectorBlock2 + freq2 \times freq2
25: end for
26: return Sim(VectorBlock1, VectorBlock2) \left( \frac{\sqrt{\text{VectorBlock1}} \times \sqrt{\text{VectorBlock2}}} {\text{VectorBlock1,2} \times 100} \right)
27: end procedure
3.3 Similarity Measures

We use two similarity measures: 1) Levenshtein similarity and 2) Cosine similarity in Step 4 of our approach discussed in Section 3.2. We use these two metrics for detecting Type-III clones.

3.3.1 Levenshtein Similarity

Levenshtein distance is named after the Russian scientist Vladimir Levenshtein, who proposed this algorithm [66]. 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., \((B_1 \text{ and } B_2)\) are candidate Type-III clone pairs.

\[
\text{Similarity} = 1 - \frac{\text{LevDist}(B_i, B_j)}{\max(\text{Len}(B_i), \text{Len}(B_j))} \times 100
\]  (3.1)

where \(\text{LevDist}\) 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 \(B_j\).

3.3.2 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
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>

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.

$$\text{CosSim}(v_1, v_2) = \cos(\alpha) = \frac{\sum_i^n v_{1i} \times v_{2i}}{\sqrt{\sum_i^n v_{1i}^2} \times \sqrt{\sum_i^n v_{2i}^2}}$$

(3.2)

where $\text{CosSim}$ is the cosine similarity between two vectors $v_1$ and $v_2$ and $\alpha$ is the angle between them.

### 3.4 Experiment Design

To compare our approach with other detectors in detecting Type-I and Type-II, we choose eight (CloneDr [10], LD [45], CCFinder [50], Dup [6], Duploc [28], Deckard [47], CDWS [77], 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 [72], CCFinder [50], Dup [6], Duploc [28], Nicad [93], Deckard [47], and CDSW [77]) detectors and use results reported by Murakami et al. [77]. 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]:

\[
\text{contained}(CF_1, CF_2) = \frac{|\text{lines}(CF_1) \cap \text{lines}(CF_2)|}{|\text{lines}(CF_1)|} \tag{3.3}
\]

where is $CF_1$ and $CF_2$ refer to the set of lines of code in code fragment $CF$. 
We also use the metric \textit{ok} value to indicate whether a candidate subsumes a reference.

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

where is \textit{CP}.\textit{CF}_1 and \textit{CP}.\textit{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 \textit{good} value metric is defined by Bellon et al. [11], to indicate whether a candidate sufficiently matches a reference. The \textit{good} value metric is stronger than the \textit{ok} value metric [11]. However, we only use the \textit{ok} value metric because it is enough to detect Type-I,II and III clones. We say \textit{CP}_1, \textit{CP}_2 are clones of each other if the \textit{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 \textit{D}. \textit{Cands} refers to a set of clone pairs, \textit{Refs} refers to the set of the clone references and \textit{DetectedRefs} refers to the set of the clone candidates. The following formulas define \textit{precision}, \textit{recall}, and \textit{F-measure}.

\[
\text{Precision}(S,D) = \frac{|\text{DetectedRef}(S,D)|}{|\text{Cands}(S,D)|}
\]

\[
\text{Recall}(S,D) = \frac{|\text{DetectedRef}(S,D)|}{|\text{Refs}(S)|}
\]

\[
\text{F-measure}(S,D) = \frac{2 \times \text{Precision}(S,D) \times \text{Recall}(S,D)}{\text{Precision}(S,D) + \text{Recall}(S,D)}
\]
\[
Recall(S,D) = \frac{|DetectedRef(S,D)|}{|Refs(S)|}
\]  
(3.6)

\[
F\text{-measure} = \frac{2 \times Recall \times Precision}{Recall + Precision}
\]  
(3.7)

3.5 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 [76].

3.5.1 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 Step1 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.

\(^2\)http://sdl.ist.osaka-u.ac.jp/~h-murakm/2014_clone_references_with_gaps/
\(^3\)https://github.com/k-hotta/ECTEC
Figure 3.2: The results of Type-I and Type-II
Figure 3.3: The results of Type-III
Figure 3.4: The results of Type-I, II, and III
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.

### 3.5.2 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 [77] 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. Because of the value precision of our approach using Levenshtien 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 is not the lowest recall. Figure 3.4b shows the value of precision. Our approach using 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 forth 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 based in vary thresholds to detect clones in the
dataset. Our approach using Cosine similarity could 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 based in vary thresholds to detect clones in the dataset. Our approach using Levenshtein distance could detect clones in a several seconds to about 15 minutes. Therefore, our approach could detect clones in a short time.
Table 3.2: Comparison the median and the average of our approaches for Type-III clones results with other existing approaches. The existing detectors results are obtained from Murakami et al. [77]. 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><strong>100.00</strong></td>
<td><strong>79.40</strong></td>
<td><strong>73.40</strong></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 LeveDist.</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>56.85</td>
<td>56.22</td>
</tr>
</tbody>
</table>

Table 3.3: Comparison the median 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. [77]. 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>21.60</td>
<td>71.20</td>
</tr>
<tr>
<td>CLAN</td>
<td>56.10</td>
<td>37</td>
<td>38.43</td>
</tr>
<tr>
<td>CCFinder</td>
<td><strong>100.00</strong></td>
<td><strong>83.95</strong></td>
<td><strong>85.03</strong></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</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>40.20</td>
<td>47.35</td>
</tr>
<tr>
<td>CDSW</td>
<td>70.63</td>
<td>51.93</td>
<td>55.29</td>
</tr>
<tr>
<td>Our Approach using LeveDist.</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>

3.6 Discussions

3.6.1 Clone References

It is hard to manually 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 [76], 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.

3.6.2 Hashing Collision

We use hash values, which as mentioned earlier, are computed using hash-code() 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.

3.6.3 Different Programming Languages of 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 can be performed more experiments in C and other programming languages to judge how our approach extends to them.

3.6.4 Thresholds for Leveshtien 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 of threshold is threshold $\tau \geq 60\%$ for Levenshtein distance and threshold $\tau \geq 80\%$ for Cosine similarity. We compare the median and average 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.

### 3.7 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
Figure 3.8: How detection of Type-III clones changes as we change Cosine similarity threshold between 35% \( \leq \tau < 100\% \).

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. Levenshtein distance is a string 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.
Acknowledgments

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). IEEE, 2015.
Chapter 4

MACHINE LEARNING FRAMEWORK FOR DETECTING SEMANTIC CODE CLONES

4.1 Introduction

In the software engineering life cycle, maintenance is the most expensive and time-consuming phase. In a large software system, pairs of segments often occur in different locations 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 cones (CC). Research has reported that 7%-23% of large software projects are code clones [8], [92]. Many studies show that a software system with frequent occurrence of code clones is difficult to maintain [91]. One of the problems with code cloning is when an
original code block, which is cloned, contains a bug, causing ripple effects to 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 [116] and code obfuscation [98], detection of malicious software [31], discovery of context-based inconsistencies [48] 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 codes. 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 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 of method blocks is highlighted in Section 4.2. In Section 4.3, we propose a new machine learning framework for detection of semantic code clones. We evaluate and compare our proposed method and reported in Section 4.4. Finally, we conclude our work in Section 4.5.

### 4.2 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, $S_i, i = 1, \ldots, M$, comprising of programming language specific executable statements such as loops, logical statements and arithmatic expressions:*

$$B = < S_1, \ldots, S_M > .$$

**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:*

$$C = \{B_1, \ldots, B_N\} .$$
**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 $B'_i = \text{trim}(B_i)$ 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) $|B'_i| = |B'_j|$, i.e., they are both of the same length after trimming, and ii) $\forall k, k = 1, \ldots, |B'_i| S'_{ik} \equiv S'_{jk}$ 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 $B'^n_i$ and $B'^n_j$ be two trimmed and normalized blocks: $B'^n_i = \text{norm}(\text{trim}(B_i))$ and $B'^n_j = \text{norm}(\text{trim}(B_j))$ 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) $|B'^n_i| = |B'^n_j|$, i.e., they are both of the same length after trimming and normalizing, and ii) $\forall k, k = 1, \ldots, |B'^n_i| S'^n_{ik} = S'^n_{jk}$.

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 versions, as described earlier. Two gapped sequences can be aligned using various techniques that generate an alignment score ($ascore$) for each alignment [29, 81]. The value of $ascore$ 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 $ascore(B_i^n, B_j^n) > \theta$ 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 [71]. In such a situation, we can approximate a version of gapped similarity ($ascore'$) between blocks $B_i^n$ and $B_j^n$ as follows:

$$ascore'(B_i^n, B_j^n) = \frac{B_i^n \cap B_j^n}{B_i^n \cup B_j^n},$$

(4.2)

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 $ascore'(B_i^n, B_j^n) > \theta$ for a user-defined threshold $\theta$. By definition, $1 \geq ascore'(.,.) \geq 0$.

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 [61], 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*

\[
\text{semsim}(B_i, B_j) = \text{semsim}(B^n_i, B^n_j) > \theta, \tag{4.3}
\]

*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 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, 111]. 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

\[
\text{semsim}(B_i^n, B_j^n) = \text{semsim}(\llbracket B_i^n \rrbracket, \llbracket B_j^n \rrbracket) \geq \theta,
\]

(4.4)

where \( \llbracket B \rrbracket \) 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 Type-II, 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.
4.3 Machine Learning in 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 some 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, $\text{features}(<B_i, B_j>) = <f_1, f_2, \cdots, f_M, C_k>$ 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 $B_i^n$ is described as given below:

$$[B_i^n] \approx <f_{i1}, \cdots, f_{ik}>.$$  \hspace{1cm} (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:

$$\text{semsim}(B_i^n, B_j^n) = \text{semsim}(<f_{i1}, \cdots, f_{ik}>, <f_{j1}, \cdots, f_{jk}>) > \theta.$$  \hspace{1cm} (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. Next, we discuss our scheme for feature generations.

4.3.1 AST and PDG: Novel features for clone detection

We pre-process the blocks by trimming and normalizing as discussed earlier. We extract basic characteristics, termed popularly as Traditional Features, like Lines of Code (LOC), number of keywords, variables, assignments, conditional statements and iteration statements [53] 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 a 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. The edge of a control flow graph can be used to detect consecutive program nodes as code clones even if they have no data or control dependency. 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. 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 between them up to level three and use them as our PDG or semantic features. For example, the feature \textit{Expr Assign Decl}, captures the number of dependency relationships that occur in sequentially in order as Expression, followed by Assignment and then followed by Declaration statements in the given code.
In Algorithm 6 we describe the feature extraction scheme. $\mathcal{L}_{\text{AST}}$ and $\mathcal{L}_{\text{PDG}}$ are the list of pre-specified AST attributes and PDG attributes (please refer Supplementary material for details) and count their frequencies in the post-order sequence of the tokens (non-leaves) extracted by PostOrderTokens and stored in $\mathcal{V}$. We avoid leaf tokens nodes as leaf nodes of all AST contain only variables. Frequencies of AST and PDG attributes are stored as features in in a vector $F$. In case the AST feature’s MatchToken matches each pre-specified AST attribute, we increase the count of that attribute or feature. The method DependencyFreq checks for the occurrence of the PDG attribute $\mathcal{L}_{\text{PDG}}$, in vector $\mathcal{V}$ and returns the frequency of such relationship in $\mathcal{V}$. Please refer to Supplementary materials 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 on other kinds of declaration, expression, and assignment, which are defined as hierarchical ordering observed in the PDG. We also extract control dependency features that are counted 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. $i < \text{count}$, for, while, if, switch etc.), obtained by counting the occurrence of the declarations which are dependent on control statements; the number of times a nested iteration occurs; number of times a nested selection occurs, and so on.
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: \textbf{INPUT} : $B$ // Target method block
2: \textbf{OUTPUT} : $F = \{f_{\text{AST}1}, \cdots, f_{\text{AST}N}, f_{\text{PDG}1}, \cdots, f_{\text{PDG}M}\}$ // Set of $N$ AST and $M$ PDG features
3: \textbf{Steps} :
4: $T \leftarrow \phi$ // AST root node
5: $L_{\text{AST}} = \{A_1 \cdots A_N\}$ // List of $N$ AST attributes
6: $L_{\text{PDG}} = \{P_1 \cdots P_M\}$ // List of $M$ PDG attributes
7: $T \leftarrow \text{GenerateAST}(B)$ // Invoking AST generator on $B$
8: $V \leftarrow \text{PostOrderTokens}(T)$ // Store post order sequence of non-leaf nodes in vector $V$
9: \text{ //Counting frequency of AST features}
10: \textbf{for} $i = 1 \cdots |L_{\text{AST}}|$ \textbf{do}
11: \hspace{1em} \textbf{for} $j = 1 \cdots |V|$ \textbf{do}
12: \hspace{2em} \textbf{if} \text{MatchToken}(A_i, V_j) \text{ then}
13: \hspace{3em} $f_{\text{AST}i} = f_{\text{AST}i} + 1$
14: \hspace{2em} \textbf{end if}
15: \textbf{end for}
16: $F = F \cup f_{\text{AST}i}$
17: \textbf{end for}
18: \text{ //Counting frequency of PDG features}
19: \textbf{for} $i = 1 \cdots |L_{\text{PDG}}|$ \textbf{do}
20: \hspace{1em} $f_{\text{PDG}i} \leftarrow \text{DependencyFreq}(P_i, V)$
21: \hspace{1em} $F = F \cup f_{\text{PDG}i}$
22: \textbf{end for}
23: \textbf{return} $F$

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 $B_i^n$ is described as given below:

$$\llbracket B_i^n \rrbracket \approx \langle f_{i1}^t, \cdots, f_{ik}^t | f_{i1}^s, \cdots, f_{ik}^s | f_{i1}^m, \cdots, f_{ik}^m \rangle.$$  \hspace{1em} (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.

4.3.2 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

$$[B_i^n] \approx features(B_i) = <f_{i1}, \cdots f_{ik}>.$$  

(4.8)
where \( k = k_l + k_s + k_m \). Similarly,

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

(4.9)

Given two blocks \( B_i \) and \( B_j \), and their clone label \( C_l \), the combined feature vector, \( \text{features}(\langle B_i, B_j \rangle) \) 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 \( 2k \). Liner combination looks like as follows:

\[
\text{features}(\langle B_i, B_j \rangle) = \langle f_{i1}, \cdots, f_{ik}, f_{j1}, \cdots, f_{jk}, C_l \rangle,
\]  

(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.

\[
\text{features}(\langle B_i, B_j \rangle) = \langle f_{i1} \times f_{j1}, \cdots, f_{ik} \times f_{jk}, C_l \rangle.
\]  

(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.
\[ \text{features}(\langle B_i, B_j \rangle) = <|f_{i1} - f_{j1}|, \ldots , |f_{ik} - f_{jk}|, C_i>. \] (4.12)

### 4.3.3 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 various pre-processing 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.

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

4.4.1 Datasets

We use IJaDataset 2.0 [102], 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 [51].
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, 96]. 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 [102]. As result, prior researchers have divided Type-III and Type-IV clones into four classes based on their syntactic similarity [96] 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.

4.4.2 Classification Models

We train and test our proposed framework using fifteen state-of-the-art classification models, starting from the popularly used Naïve Bayes [49] 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 models including probabilistic and non-probabilistic, generative and discriminative linear and non-linear, regression, decision trees and distance based models.

Naïve Bayes [49] is a simple probabilistic classifier based on applying Bayes’ rule. Linear Discriminant Analysis (LDA) [74] 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 classifica-
Table 4.1: Different classification techniques used

<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>IBK</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, work with numeric attributes allowed</td>
<td>Weka 3.8</td>
</tr>
<tr>
<td>Bootstrap 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 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>Ensembles 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>

LogitBoost [33] is a boosting classification algorithm. LogitBoost and AdaBoost are close to each other in that both perform an additive logistic regression. In addition to that we use several tree ensemble models including Extra Trees [36], Rotation Forest [90] coupled with Principal Components Analysis (PCA), Random Forest [13] and Random Committee [112]. 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 aggregate their individual predictions. We also use decision tree algorithms such as J48 [97] and Random tree [63] 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 fast and accurate boosting tree model proposed recently. The classification models used in our work are summerised in Table 4.1.

We represent pair instances as one vector as explained above. Similar blocks are detected by 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 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 and the same as in the overall dataset.

### 4.4.3 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 our features are able to achieve a high detection accuracy. Due to space limitations we report only 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 Supplementary materials we provide at (website). To generate AST from a given block in order to extract features we use Eclipse Java Development Tools (JDT).

#### 4.4.4 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
error 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. Preliminary results are available in [100].

4.4.4.1 Varying data size and feature types

We assess the importance of combining Traditional, AST and PDG features different combinations 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 performances 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.
Figure 4.6: Performance of three best classifiers with synthetic and semantic features
4.4.4.2 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 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 [84] and Information Gain [83]. 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 established the fact that our features are crucial in deriving high accuracy detection results.

Performance comparisons 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 maximum value of the reported range. Interestingly, a majority of the detection methods are incapable of detecting semantic clones or Type-IV clones. Figure 4.9 show 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 performance better with respect to all other methods in detecting Type-
Figure 4.7: Learning Curve: Performance of Random Forest and Xgboost with varying features.

I/II, VST3, and ST3 clones based on the F-measure metric. We report only results for 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.
Figure 4.8: Performance of Random Forest and Rotation Forest with varying features using Gain Ratio and InfoGain feature selection algorithms.

(a) Distance combination

(b) Multiplicative combination

(c) Linear combination
Figure 4.9: Performance comparison of different detection methods with respect to different assessment metrics.

4.5 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 using them for discussing graph isomerism. In order to avoid a-
priori bias towards any classification models, we use 15 state-of-the-art classification
model to obtain their relative performance our features. Experimental results clearly
indicate that our proposed features are highly valuable to achieve high detection ac-
curacy.

We would like to extend our work to achieve further improvements in next
Chapters, for example, by using features of Java byte codes obtained by compiling
Java programs.
Acknowledgments

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, Anaheim, California, December, 2016
Chapter 5

AUTOMATED LABELING TYPE-IV CODE CLONE USING

JAVA BYTESCODE

5.1 Introduction

Software maintenance is a most 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 [87]. 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 oracle; this detector of which 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 bytecode. 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 might 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 Section 5.2. Characteristics of Java bytecode are described in Section 5.3. In Section 5.4, the proposed method is discussed in detail. Finally, the chapter is concluded in Section 5.5.

5.2 Prior Research

Lavoie and Merlo [64] introduce clone detectors for Type-III clones using the Levenshtein metric with size up to a few MLOCs, without relying on statistical properties. The resulting clone references ae for two Java software systems, namely Tomcat and Eclipse. Their work only detects Type-III clones.

Bellon et al. [11] perform experiments on six detectors using eight large C and Java programs. Bellon’s datasets are widely used. However, Bellon’s dataset identifies only Type-I,II and III clones. Their approach, introduced in 2007, has become the standard for evaluation of every new clone detectors. The approach has three steps. 1) Eight target software systems, namely netbeans, eclipse-ant, eclipsejdtcor, J2sdk1.4.0-javax-swing, weltab, cook, snns, and postgresql are selected. 2) Code clones are detected by the state-of-the-art clone detectors. Then, the information containing locations of the detected clones are sent to Bellon is detector. 3) The ob-
servers had sent the 2% of the detected code clones randomly and checked each of these code clones manually.

Murakami et al. [76] extend Bellon’s dataset of clone references by adding information containing locations of gapped lines. Bellon’s datasets do not contain gapped lines, location information for Type-III clones. 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.

Kurtz and Le [60] obtain oracle clones from three open source systems, namely Apache, Python and PostgreSQL. They generate a set of method level semantic code clones with high confidence to assist evaluation 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 small, its size is only 66 methods clone pairs are of Type-I and 9 pairs are of Type-IV.

5.3 Java Bytecode Overview

Java bytecode is a sequence of instructions for the virtual machine to simulate basic functionalities such as conditions and loops [103]. 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.

5.4 The Proposed Method
Algorithm 5 Automated Labeling Types of Clones

1: procedure OracleClones(MethodCodes)
2:   TI/IIClones ← 0, TIIIClones ← 0,
3:   TIVClones ← 0
4:   for i ← 0, MethodCodes.Length do
5:     for j ← i+1, MethodCodes.Length do
6:       Code₁ ← CompileToByteCode(MethodBᵢ)
7:       Code₂ ← CompileToByteCode(MethodBⱼ)
8:       SimM ← LevDist(MethodBᵢ, MethodBⱼ)
9:       SimBCode ← LevDist(Code₁, Code₂)
10:      if SimM = α then
11:         if SimBCode ≥ π then
12:           TI/IIClones ← TI/IIClones + 1
13:         end if
14:       else if μ ≤ SimM && SimM > α then
15:         if SimBCode ≥ π then
16:           TIIIClones ← TIIIClones + 1
17:         end if
18:       else SimM < μ
19:         if SimBCode ≥ π then
20:           TIVClones ← TIVClones + 1
21:         end if
22:       end if
23:     end for
24:   end for
25: return TI/IIClones, TIIIClones, TIVClones
26: end procedure

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 are used. The proposed method consists 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.
Figure 5.1: A new framework for labelling semantic code clones

(a) **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.

(b) **Identify similar method blocks using Levenshtein distance.** This helps identify similar method codes using Levenshtein distance after identifying all normal-
ized codes. We choose a threshold based on their syntactic similarity as following: Type-I and II clones have syntactic similarity 100%, Type-III clones that have syntactic similarity \([50\% - 100\%]\) and Type-IV clones with similarity in \((0\%- 50\%]\); these percentages are similar to those used by prior researchers [96]. We compare each method and its Java bytecode against all other methods in the same dataset for clones.

(c) 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.

(d) 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).

(e) 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.

(f) Inspect Manually. This step is a manual inspection on all pairs of methods detected using softaware 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 Master and PhD degrees in Computer Science.
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-I V</th>
<th>False Agreement (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Suple</td>
<td>152</td>
<td>30</td>
<td>59</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>

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. The publicly available datasets. A brief summery of the extended datasets is given in Table 5.1. In the table, the second column indicates how many paired-blocks we extracted to expand the existing dataset. Agreement refers to the probability of reliability between observers or raters. We compute Kappa statistic [104] 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.

**g. 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.
The goal of this study is to analyze the source code uses bytecode is an intermediate representation instead of using any other intermediate representation. 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 selections 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 6

OBFSUCATED CODE DETECTION- AN APPLICATION OF SEMANTIC CLONE DETECTION SCHEME

6.1 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 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 invariant. 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 Section 6.2, background is discussed in detail. Prior research of code obfuscation are described in Section 6.3. Our code clone detection framework is discussed in Section 6.4.
We discuss the experiments we perform in Section 6.5. Finally, the chapter is concluded in Section 6.6.

### 6.2 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 harder. 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, 111]. 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

$$SemSim(C_i, C_j) = SemSim^*([C_i], [C_j]) > \varphi,$$  \hspace{1cm} (6.1)$$

where $SemSim^*(.,.)$ is a formal measure of similarity between the two pieces of code. $[C_i]$ is the formal semantics of code fragment $C_i$ computed following a formal approach. In this paper, we will not delve deeper into how $[C_i]$,
\([C_j]\) or, \(SemSim^*(\cdot,\cdot)\) may be computed exactly following semantic theory. In practice, we approximate the computation of \(SemSim(C_i, C_j)\) using other means as discussed in this paper. In other words, the primary focus of using this paper is an approximate computation of \(SemSim^*(\cdot,\cdot)\) using non-formal semantic means. Assuming, we can provide a good computational approximation to \(SemSim^*(\cdot,\cdot)\), we can proceed to define semantic clones. We call this approximation \(SemSim^*(\cdot,\cdot)\).

**Definition 10 (Semantic Clone).** Code \(C_i\) is a semantic clone 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 code \(C_j\) is obfuscated version of another code \(C_i\) if they exhibit similar functionality although the structurally they are different from each other. It is similar to semantic clone. In other words, obfuscated code pairs can be represented as discrete function as follows.

\[
Obfuscated(C_i, C_j) = \begin{cases} 
1, & \text{if } SemSim(C_i, C_j) > \varphi \\
SynSim(C_i, C_j) = 0; & \text{if } SemSim(C_i, C_j) = 0; \\
0, & \text{otherwise,}
\end{cases} \tag{6.2}
\]

where \(SemSim(\cdot,\cdot)\) is a semantic similarity function, and \(\varphi\) is a user set threshold.

Several effective methods have been proposed for discovering obfuscated code. A brief sketch of these methods is given below.
6.3 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 2, followed by obfuscated code detection.

Obfuscated code detection methods are abundant in the literature. Likarish et al. [68] propose an obfuscation detection scheme in JavaScript scripts. Automatic analysis of malware behavior using machine learning is proposed by Rieck et al. [88]. 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. [108] propose a technique to detect malicious JavaScript in web pages using deep features extracted by Stacked Denoising Auto-encoders (SdA) model for classification. Results are promising although SdA is slow in training. O’Kane et al. [79] present a method using an optimal set of operational codes (opcodes) from an assembly language or machine language instructions.

Ragkhitwetsagul et al. [85] study and evaluate 30 tools using two experimental scenarios for Java source code. They perform pervasive modifications of source code using the ARTIFICE tool [98] and bytecode obfuscation using the ProGuard tool ¹, which optimizes Java bytecode and provides reverse engineer-

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

6.4 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 earlier, obtain the formal semantics of the two using a method alluded to earlier; and, compare the formal semantic representations using Equation 4.3. 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 obfuscated 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 fragments 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 fea-
tures 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 about various code fragment features we use for semantic clone or obfuscated code detection.

### 6.4.1 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 bytecodes 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.
Table 6.1: Byte Code 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,</td>
</tr>
<tr>
<td></td>
<td>ifnonnull, ifnonnull, if.icmpeq,</td>
</tr>
<tr>
<td></td>
<td>if.icmpgt, if.icmpge, if.acmpne,</td>
</tr>
<tr>
<td></td>
<td>if.icmplt, if.icmple, if.icmple, if.icmple, if.icmple,</td>
</tr>
<tr>
<td></td>
<td>if.icmple, if.icmple, if.icmple, if.icmple, if.icmple,</td>
</tr>
<tr>
<td>Compound Cond. Branch</td>
<td>tableswitch, lookupswitch</td>
</tr>
<tr>
<td>Comparisons</td>
<td>lcimp, fcmpg, fcmpq, dcmpg, dcmpq</td>
</tr>
</tbody>
</table>

We attempt to represent the meaning of a program by elucidating interdependency relationships among different bytecode constructs. We represent such dependency as a graph called ByteCode Dependency Graph (BDG). An illustration of 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.

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
Table 6.2: Categorization of Byte Code 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, icontst, dconst, fconst, lconst</td>
</tr>
<tr>
<td>Arithmetic</td>
<td>iadd, dadd, fadd, ladd, isub, dsub, fsub, lsub, imul, dmul, fmul, lmul, idiv, ddiv, fdiv, ldiv, irem, drem, frem, lrem</td>
</tr>
<tr>
<td>Type Conversion</td>
<td>i2l, i2f, i2d, l2f, l2d, f2i, f2l, f2d, d2i, d2l, d2f, i2b, i2c, i2s</td>
</tr>
</tbody>
</table>

Figure 6.1: A BDG showing control and data dependency among the instructions.

fragments. We pre-process 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, 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 DependencyFreq checks for all possible control dependency relationships that persist in the bytecode. Similar to 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 $L_{BDG_i}$. Frequencies of BC and BDG attributes are finally stored into a vector $F$ as features of a given Java program. Please refer to Supplementary materials for the details of the features we extract as BC and BDG features.

### 6.4.2 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
Algorithm 6 ByteCode & BDG Feature Extraction

1: **INPUT**: C // Java Source Code
2: **OUTPUT**: \( F = \{ f_{BC_1}, \cdots, f_{BC_N}, f_{BDG_1}, \cdots, f_{BDG_M} \} \) // Set of N BC and M BDG features
3: **Steps**: 
4: \( \mathcal{L}_{BC} = \{ B_1 \cdots B_N \} \) // List of N BC attributes
5: \( \mathcal{L}_{BDG} = \{ D_1 \cdots D_M \} \) // 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 \( i = 1 \cdots |\mathcal{L}_{BC}| \) do
9: \hfill for \( j = 1 \cdots |V| \) do
10: \hfill \hfill if \( \text{MatchToken}(B_i, V_j) \) then
11: \hfill \hfill \hfill \hfill \hfill f_{BC_i} = f_{BC_i} + 1
12: \hfill \hfill \hfill \hfill \hfill end if
13: \hfill \hfill end for
14: \hfill \hfill \hfill \hfill \hfill F = F \cup f_{BC_i}
15: \hfill \hfill end for
16: \hfill //Counting frequency of BDG features
17: for \( i = 1 \cdots |\mathcal{L}_{BDG}| \) do
18: \hfill \hfill \hfill \hfill \hfill F_{BDG_i} \leftarrow \text{DependencyFreq}(D_i, V)
19: \hfill \hfill \hfill \hfill \hfill F = F \cup F_{BDG_i}
20: \hfill \hfill end for
21: \hfill \hfill \hfill return \( F \)

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 meanings, 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 \(^2\) we use. We traverse
the AST in post-order manner and extract only non-leaf nodes containing pro-
gramming 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 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 ex-

---

\(^2\)http://www.eclipse.org/jdt/
trol (e.g. i < count, for, while, if, switch etc.), the numbers of times nested
iterations occur; the numbers of times nested selections occur, and so on.

In addition to all of the features discussed above, we extract basic source
code characteristics, termed popularly as Traditional Features. They include
numbers of Lines of Code (LOC), numbers of keywords, variables, assign-
ments, conditional statements and iteration statements [53] used in a given piece
of source code.

6.4.3 Fusion of Code Features

We combine feature vectors (Equation 6.3) extracted from a pair of tar-
get and reference code codes to create the training dataset, which is similar to
Subsection 4.3.2.

$$[C_i^n] \approx <f_{i1}^l, \ldots f_{ik_b}^l \mid f_{i1}^t, \ldots f_{ik_a}^t \mid f_{ik_d}^d \mid f_{ik_h}^h \mid f_{ik_p}^p>.$$  

(6.3)

In this equation, we denote the different categories of features with dif-
ferent superscripts: $l$ for low level or byte code related features and $h$ for high
level source code features. We denote the different types of features with dif-
ferent 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.
We fuse the sequence of features from the two different codes. Although there are five type of features covering both low and high level features in the description of a code fragment, to simplify the notation, we can rewrite Equation 6.4 and 6.5, without distinguishing among the feature types, as:

\[
\begin{align*}
\mathbb{[C^\text{a}_i]} &= \text{features}(C_i) = \langle f_{i1}, \ldots, f_{ik} \rangle, \\
\mathbb{[C^\text{a}_j]} &= \text{features}(C_j) = \langle f_{j1}, \ldots, f_{jk} \rangle.
\end{align*}
\]

(6.4)

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

\[
\begin{align*}
\mathbb{[C^\text{a}_j]} &= \text{features}(C_j) = \langle f_{j1}, \ldots, f_{jk} \rangle.
\end{align*}
\]

(6.5)

We use known pair of cloned or obfuscated codes for feature extraction and label the class of the feature vector as true clone or 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, \( \text{features}(\langle C_i, C_j \rangle) \) can now be represented as a fused feature vector. We fuse the two vectors in three different ways as discussed in Section 4.3.2.

### 6.4.4 Code Similarity as a Feature

We use the text similarity score of 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
Figure 6.2: Share of categories of features used.

bytecodes $B_i$ and $B_j$ respectively. We tokenize each source code fragments and count frequencies of the tokens. We calculate cosine similarity [15] using the frequency vectors as follows.

\[
S_1(C_i, C_j) = \cos(\theta) = \frac{\sum_{i,j} c_i \times c_j}{\sqrt{\sum_i c_i^2} \times \sqrt{\sum_j c_j^2}} \quad (6.6)
\]

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 of two Java bytecode fragments, $S_2(B_i, B_j)$ using the above equation.

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

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

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.
6.4.5 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 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.

6.5 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 semantic clone as well as obfuscated code detection accuracy. We use an ensemble of selected classifiers and compare the effectiveness of the proposed framework with the state-of-the-art clone and obfuscated code detection methods.
6.5.1 Datasets

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

6.5.1.1 Clone Dataset

We use six Java code clone and three obfuscated code datasets for training and testing. Details of the datasets are reported 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-javawing*, *eclipse-jdtcore*, *EIRC* and *Suple*.

6.5.1.2 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 Artifice [98]. Then, the original and obfuscated files are compiled to bytecode. Both byte code files are obfuscated further using ProGuard to create stronger obfuscation. After that, all four bytecode files are decompiled using either Krakatau ³ or Procyon ⁴ giving back eight additional obfuscated source code files [85]. We generate nine pervasively modified version of each original source code, resulting in a total 50 of files for the dataset [85].

We select PacMan game ⁵ as our second subject system [98]. It contains of 21 files and 2400 lines of code. The classes are further transformed using renaming, contraction, expansion, loop transformations [98].

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 content 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 obfuscated source code files. We obtain 785 Java bytecode files for the dataset. A summary of the dataset is given in Table 6.3.

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/
6.5.2 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 [49], LibLinear SVM [30], Instance Based Learner (IBK) [3],
Bagging [12], Logit Boost [33], Random Committee [112], Random Subspace [42], Rotation Forest [90], J48 [97], and Random Forest [13] classifiers and ensemble them based on majority decisions to obtain the final class label.

6.5.3 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.

6.5.3.1 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 features 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 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 adopted. However, for the Algorithm dataset, linear fusion gives better results in comparison to other methods.

### 6.5.3.2 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 using category wise selection of features we use a random forests based feature selection algorithm, namely Mean
Figure 6.8: Effectiveness of the framework on detecting obfuscated code using features fusions.

Figure 6.9: Learning Curve: Performance of ensemble classifier on clone dataset with varying number of features.
Decrease Impurity (MDI) [70] 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 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.
6.5.4 Performance comparison

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

6.5.4.1 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 [77] and [44] a range of F-scores for detecting Type I, Type II and III. 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 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 a reliable future error prediction, we build a model we need to evaluate it in order to compare it with another model or in order to estimate predictive performance of the model. We need to evaluate the model on a different, independent and identically distributed sets that we have used for building the model. We integrate j2sdk14.0-javax-swing and eclipse-jdtcore systems and use them as training set and building the model in order to estimate the parameters of the model during the learning phase such as in 6.4. Then, we use a different system, which is completely new data, for a model evaluation.

In terms of accuracy, we build our model using distance combination and Traditional, AST, PDG, BC and BDG features and report the results such as in Figure 6.4. The performance of ensemble classifier using all features, which are (Traditional, AST, PDG, BC and BDG) features, produces better results than the performance of ensemble classifier using only AST and PDG features.

### 6.5.4.2 Performances of Obfuscated Code Detectors

We use the previously reported performance scores [85] in terms of Precision, Recall and F1-score values. We consider the maximum value of the range reported by other authors once again; 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
Detection rates for Type-I and II clones

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

(b) Detection rates for Type-III clones

Figure 6.12: Prediction effectiveness of proposed framework in comparison to state-of-the-art clone detectors in terms of F-Score

obfuscated code. Our method is the winner with the highest F1-Score (100%) in all cases of detecting obfuscated codes based on three different datasets, Ob-SCode, ObsCode*(karkatau), and ObsCode* (procyon). ObsCode*(karkatau) and ObsCode* (procyon) are the variation 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 adopt. In terms of accuracy, our method is the best compared to other four methods, which are Text-based (JPLAG(v.2.2.1)), Token-
Figure 6.13: Effectiveness of various obfuscated code detection tools on ObsCode dataset
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 of obfuscations types, viz, contraction, expansion, loop transformation, renaming, our model is the winner with 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.

We also build a model for detecting code obfuscation using three ways for avoiding error prediction. 1) The model is built using *obsCode+Algorithms* sets and evaluate the model using *PacMan*. 2) The model is built using *obsCode+PacMan* sets and evaluate the model using *Algorithms*. 3) The model is built using *PacMan+Algorithms* sets and evaluate the model using *ObsCode* such as in 6.6. In terms of accuracy, our obfuscation code detection model is still even producing better results when the model is built with a sort of dataset and test the same model with another.

6.6 Chapter Summary

The semantics of a program written by anonymous programmer is difficult to characterize, especially for detecting software clones and obfuscated programs. A number of methods and software tools are available for 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
Figure 6.14: Tool performance comparison on the PacMan data in terms of accuracy, which original program compared to obfuscated programs for each clone detection technique. The obfuscation are abbreviated as follows: C–contraction, E–expansion, L–loop transformation, R–renaming [98]

<table>
<thead>
<tr>
<th>OBfuscated Type</th>
<th>Our Method (Distance)</th>
<th>Our Method (Multiple)</th>
<th>Our Method (Linear)</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EL</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>L</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LC</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RE</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RL</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RC</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>REC</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RCL</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Semantics of program codes using low and high level program features derived from bytecode, the data and conditional dependency with in the bytecode, AST and PDG. We perform an extensive set of experiments to show that our framework is capable to detect both code clone and obfuscated code equally well. The detailed results we present in this paper and in Supplementary Materials clearly establish 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 Java code. We are exploring ways to make the framework more general in
nature, with an eye to its high commercial importance.
Chapter 7

CONCLUSION AND FUTURE WORK

7.1 Conclusion

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

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

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

Majority of the publicly available datasets lacking 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 us-
ing Java bytecode similarity to label publicly available datasets, namely, *Su-
ples*, *netbeans-javadoc*, *eclipse-ant*, *EIRC*, *j2sdk14.0-javax-swing*, and *eclipse-
* *jdtcore*.

Finally, we reported a potential application of our proposed clone detection framework for detecting obfuscated codes. An integrated framework for detecting both Java code clones and obfuscated codes is 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 use an ensemble of state-of-the-art classification models to evaluate the effectiveness of our proposed idea.

Experimentally all the proposed methods are evaluated and compared with state-of-the-art methods/tools in the light of various publicly available clone and obfuscated datasets. Results reveals that our proposed methods are superior in performance in detecting both code clones and obfuscated codes.

### 7.2 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 codes. A future endeavor for developing a general framework covering all major programming languages will have better commercial value.
Semi-supervised learning has been focused on designing algorithms to effectively exploit the unlabeled data. It also can use a small amounts of labeled data for helping to label unlabeled data and reduce the costs associated with labeling. We plan to construct clone references for Type-IV clones using semi-supervised learning algorithms and therefore tend to be more efficiency in identify semantic clones between different programming languages for helping researchers evaluate future code clone detection techniques.

Convolutional Neural Network (CNN or ConvNet) [59] is a type of feed-forward artificial neural network which the connectivity pattern between its neurons. We are going to present a novel ConvNet model for detecting Java code clones and obfuscated codes 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 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 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, and using code block pairs as 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.

We have presented several approaches from simple to complex for detecting code clones and obfuscated code. We have also compared our methods against existing methods to find advantages and disadvantages points in different approaches in Chapter 2.
References


[16] Tianqi Chen and Carlos Guestrin. Xgboost: A scalable tree boosting


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


[40] Yoshiki Higo, Ken-ichi Sawa, and Shinji Kusumoto. Problematic code 

[41] Yoshiki Higo, Ueda Yasushi, Minoru Nishino, and Shinji Kusumoto. In-

[42] Tin Kam Ho. The random subspace method for constructing decision 
forests. *IEEE transactions on pattern analysis and machine intelligence*, 

[43] Susan Horwitz, Jan Prins, and Thomas Reps. On the adequacy of pro-
gram dependence graphs for representing programs. In *Proceedings of 
the 15th ACM SIGPLAN-SIGACT symposium on Principles of program-

accurate is coarse-grained clone detection?: Comparision with fine-

[45] Benjamin Hummel, Elmar Juergens, Lars Heinemann, and Michael Con-
radt. Index-based code clone detection: incremental, distributed, scalable. In *Software Maintenance (ICSM), 2010 IEEE International Con-


[67] Zhenmin Li, Shan Lu, Suvda Myagmar, and Yuanyuan Zhou. Cp-miner:


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


[81] Pavel Pevzner. *Computational molecular biology: an algorithmic ap-


[113] Xifeng Yan, Jiawei Han, and Ramin Afshar. Clospan: Mining closed sequential patterns in large datasets. In In SDM, pages 166–177, 2003.


## FEATURES AND RESULTS OF CHAPTER 4

### A.1 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 block.</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 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) Block&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 ] Block [ CatchClause ] finally Block ;&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 Statements</td>
<td>Counts &quot;SingleVariableDeclaration and 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 Statements</td>
<td>Counts &quot;TypeDeclaration and EnumDeclaration&quot; in method blocks.</td>
</tr>
<tr>
<td></td>
<td>No. of Type Parameters</td>
<td>Counts &quot;ExtendedModifier Identifier [ extends Type &amp; Type ] ;&quot; in method blocks.</td>
</tr>
</tbody>
</table>
## A.2 AST Features

### Table A.2: AST features

<table>
<thead>
<tr>
<th>Category</th>
<th>Features</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AST</td>
<td>No. of Class Instance Creations</td>
<td>Counts &quot;Expression : new Name (Expression , Expression )&quot; in method blocks.</td>
</tr>
<tr>
<td></td>
<td>No. of Array Creations</td>
<td>Counts &quot;new PrimitiveType [Expression ]</td>
</tr>
<tr>
<td></td>
<td>No. of Array Initializer</td>
<td>Counts &quot;Expression , Expression []&quot; in method blocks.</td>
</tr>
<tr>
<td></td>
<td>No. of Array Access</td>
<td>Counts &quot;Expression [Expression []&quot; in method blocks.</td>
</tr>
<tr>
<td></td>
<td>No. of Cast Expressions</td>
<td>Counts &quot;(Type ) Expression&quot; in method blocks.</td>
</tr>
<tr>
<td></td>
<td>No. of Constructor Invocations</td>
<td>Counts &quot;&lt;Type , Type &gt; this (Expression , Expression )&quot; in method blocks.</td>
</tr>
<tr>
<td></td>
<td>No. of Field Access</td>
<td>Counts &quot;Expression . Identifier&quot; in method blocks.</td>
</tr>
<tr>
<td></td>
<td>No. of Field Declarations</td>
<td>Counts &quot;[Javadoc] ExtendedModifier Type VariableDeclarationFragment , VariableDeclarationFragment &quot; in method blocks.</td>
</tr>
<tr>
<td></td>
<td>No. of Super Method Invocations</td>
<td>Counts &quot;ClassName . super . Identifier (Expression , Expression )&quot; in method blocks.</td>
</tr>
<tr>
<td></td>
<td>No. of Array Type</td>
<td>Counts &quot;Type Dimension Dimension&quot; in method blocks.</td>
</tr>
<tr>
<td></td>
<td>No. of Assert Statement</td>
<td>Counts &quot;assert Expression : Expression &quot; in method blocks.</td>
</tr>
<tr>
<td></td>
<td>No. of Boolean Literal</td>
<td>Counts &quot;true, false&quot; in method blocks.</td>
</tr>
<tr>
<td></td>
<td>No. of Break Statement</td>
<td>Counts &quot;break [Identifier ] ;&quot; in method blocks</td>
</tr>
<tr>
<td></td>
<td>No. of Continue Statement</td>
<td>Counts &quot;continue [Identifier ] ;&quot; in method blocks.</td>
</tr>
<tr>
<td></td>
<td>No. of Enum Declaration</td>
<td>Counts &quot;[Javadoc] ExtendedModifier enum Identifier (implements Type , Type )</td>
</tr>
<tr>
<td></td>
<td>No. of Parameterized Type</td>
<td>Counts &quot;Type &lt;Type , Type &gt;&quot; in method blocks.</td>
</tr>
<tr>
<td></td>
<td>No. of Super Field Access</td>
<td>Counts &quot;ClassName . super . Identifier&quot; in method blocks.</td>
</tr>
<tr>
<td></td>
<td>No. of This Expression</td>
<td>Counts &quot;ClassName . this&quot; in method blocks</td>
</tr>
<tr>
<td></td>
<td>No. of Throw Statement</td>
<td>Counts &quot;throw Expression ;&quot; in method blocks</td>
</tr>
<tr>
<td></td>
<td>No. of Variable Declaration Expression</td>
<td>Counts &quot;ExtendedModifier Type VariableDeclarationFragment , VariableDeclarationFragment &quot; in method blocks</td>
</tr>
<tr>
<td></td>
<td>No. of Block</td>
<td>Counts &quot;Statement ;&quot; in method blocks.</td>
</tr>
<tr>
<td></td>
<td>No. of Catch Clause</td>
<td>Counts &quot;Expression InfixOperator Expression InfixOperator Expression&quot; in method blocks.</td>
</tr>
<tr>
<td></td>
<td>No. of Infix Expressions</td>
<td>Counts &quot;Expression instanceof Type&quot; in method blocks.</td>
</tr>
<tr>
<td></td>
<td>No. of Instance of Expression</td>
<td>Counts &quot;Type &amp; Type &amp; Type&quot; in method blocks.</td>
</tr>
<tr>
<td></td>
<td>No. of Intersection Type</td>
<td>Counts &quot;Type &amp; Type &amp; Type&quot; in method blocks.</td>
</tr>
<tr>
<td></td>
<td>No. of Labeled Statement</td>
<td>Counts &quot;Identifier : Statement&quot; in method blocks.</td>
</tr>
<tr>
<td></td>
<td>No. of Member Value Pair</td>
<td>Counts &quot;SimpleNameExpression&quot; in method blocks.</td>
</tr>
<tr>
<td></td>
<td>No. of Method Invocations</td>
<td>Counts &quot;Expression . &lt;Type , Type &gt; Identifier (Expression , Expression )&quot; in method blocks.</td>
</tr>
<tr>
<td></td>
<td>No. of Method Refs</td>
<td>Counts &quot;Name # Identifier (MethodRefParameter — , MethodRefParameter )&quot; in method blocks.</td>
</tr>
<tr>
<td></td>
<td>No. of Method Ref Parameter</td>
<td>Counts &quot;Type [ ... ] Identifier&quot; in method blocks.</td>
</tr>
<tr>
<td></td>
<td>No. of Parenthesized Expressions</td>
<td>Counts &quot;Expression /&quot; in method blocks.</td>
</tr>
<tr>
<td></td>
<td>No. of Prefix Expression</td>
<td>Counts &quot;PrefixOperator Expression&quot; in method blocks.</td>
</tr>
<tr>
<td></td>
<td>No. of Primitive Types</td>
<td>Counts &quot;Annotation f type or (Annotation ) short or (Annotation ) char or (Annotation ) int or (Annotation ) long or (Annotation ) float or (Annotation ) double or (Annotation ) boolean or (Annotation ) void in method blocks.</td>
</tr>
<tr>
<td></td>
<td>No. of Qualified Name</td>
<td>Counts &quot;Name . SimpleName&quot; in method blocks.</td>
</tr>
<tr>
<td></td>
<td>No. of Qualified Type</td>
<td>Counts &quot;Type . Annotation SimpleName&quot; in method blocks.</td>
</tr>
<tr>
<td></td>
<td>No. of Simple Names</td>
<td>Counts &quot;Identifier&quot; in method blocks.</td>
</tr>
<tr>
<td></td>
<td>No. of Simple Types</td>
<td>Counts &quot;Annotation , TypeName&quot; in method blocks.</td>
</tr>
<tr>
<td></td>
<td>No. of Type Declaration</td>
<td>Counts &quot;[Javadoc] ExtendedModifier class Identifier (&lt;TypeParameter , TypeParameter &gt;</td>
</tr>
<tr>
<td></td>
<td>No. of Type Literal</td>
<td>Counts &quot;Type — void , class&quot; in method blocks.</td>
</tr>
<tr>
<td></td>
<td>No. of Union Type</td>
<td>Counts &quot;&quot; in method blocks.</td>
</tr>
<tr>
<td></td>
<td>No. of Wild card Types</td>
<td>Counts &quot;Annotation , [ extends — super ] Type&quot; in method blocks.</td>
</tr>
</tbody>
</table>
### A.3 PDG Features

Table A.3: PDG features. “→” means “dependent”

<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 → Control in method blocks.</td>
</tr>
<tr>
<td></td>
<td>No. of AR</td>
<td>Counts Assignment → Return in method blocks.</td>
</tr>
<tr>
<td></td>
<td>No. of EC</td>
<td>Counts Expression → Control in method blocks.</td>
</tr>
<tr>
<td></td>
<td>No. of DA</td>
<td>Counts Declaration → Assignment in method blocks.</td>
</tr>
<tr>
<td></td>
<td>No. of ACD</td>
<td>Counts Assignment → Control → Declaration in method blocks.</td>
</tr>
<tr>
<td></td>
<td>No. of CWA</td>
<td>Counts Control While statement → Assignment in method blocks.</td>
</tr>
<tr>
<td></td>
<td>No. of CFA</td>
<td>Counts Control For statement → Assignment in method blocks.</td>
</tr>
<tr>
<td></td>
<td>No. of CDA</td>
<td>Counts Control → Declaration → Assignment in method blocks.</td>
</tr>
<tr>
<td></td>
<td>No. of CIA</td>
<td>Counts Control If → 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 → Declaration in method blocks.</td>
</tr>
<tr>
<td></td>
<td>No. of CWD</td>
<td>Counts Control while → Declaration in method blocks.</td>
</tr>
<tr>
<td></td>
<td>No. of CFH</td>
<td>Counts Control For Statement → Declaration in method blocks.</td>
</tr>
<tr>
<td></td>
<td>No. of CID</td>
<td>Counts Control If statement → Declaration in method blocks.</td>
</tr>
<tr>
<td></td>
<td>No. of CSD</td>
<td>Counts Control Selection → Declaration in method blocks.</td>
</tr>
<tr>
<td></td>
<td>No. of ED</td>
<td>Counts Expression → Declaration in method blocks.</td>
</tr>
<tr>
<td></td>
<td>No. of ECD</td>
<td>Counts Expression → Control → Declaration in method blocks.</td>
</tr>
<tr>
<td></td>
<td>No. of EAD</td>
<td>Counts Expression → Assignment → Declaration in method blocks.</td>
</tr>
<tr>
<td></td>
<td>No. of CWAEx</td>
<td>Counts Control While statement → Expression in method blocks.</td>
</tr>
<tr>
<td></td>
<td>No. of CFAEx</td>
<td>Counts Control For statement → Expression in method blocks.</td>
</tr>
<tr>
<td></td>
<td>No. of CDEx</td>
<td>Counts Control → Declaration → Expression in method blocks.</td>
</tr>
<tr>
<td></td>
<td>No. of CIEX</td>
<td>Counts Control If statement → Expression in method blocks.</td>
</tr>
<tr>
<td></td>
<td>No. of CIES</td>
<td>Counts Control Selection → Expression in method blocks.</td>
</tr>
<tr>
<td></td>
<td>No. of IFIF</td>
<td>Counts If statement → If statement in method blocks.</td>
</tr>
<tr>
<td></td>
<td>No. of IFFOR</td>
<td>Counts If statement → For statement in method blocks.</td>
</tr>
<tr>
<td></td>
<td>No. of IFIFSWITCH</td>
<td>Counts If statement → Switch statement in method blocks.</td>
</tr>
<tr>
<td></td>
<td>No. of IFWHILE</td>
<td>Counts If statement → While Statement in method blocks.</td>
</tr>
<tr>
<td></td>
<td>No. of IFDOWILE</td>
<td>Counts If statement → DoWhile Statement in method blocks.</td>
</tr>
<tr>
<td></td>
<td>No. of FORIF</td>
<td>Counts For statement → If Statement in method blocks.</td>
</tr>
<tr>
<td></td>
<td>No. of FORFOR</td>
<td>Counts For statement → For Statement in method blocks.</td>
</tr>
<tr>
<td></td>
<td>No. of FORSWITCH</td>
<td>Counts For statement → Switch Statement in method blocks.</td>
</tr>
<tr>
<td></td>
<td>No. of FORWHILE</td>
<td>Counts For statement → While Statement in method blocks.</td>
</tr>
<tr>
<td></td>
<td>No. of FORDOWILE</td>
<td>Counts in method blocks.</td>
</tr>
<tr>
<td></td>
<td>No. of IterationIteration</td>
<td>Counts Iteration statement → Iteration Statement in method blocks.</td>
</tr>
<tr>
<td></td>
<td>No. of SelectionIteration</td>
<td>Counts Selection statement → Iteration Statement in method blocks.</td>
</tr>
<tr>
<td></td>
<td>No. of SelectionReturn</td>
<td>Counts Selection statement → Return Statement in method blocks.</td>
</tr>
<tr>
<td></td>
<td>No. of IterationReturn</td>
<td>Counts Iteration statement → Return Statement in method blocks.</td>
</tr>
</tbody>
</table>

- Nested Loop: This feature collects the number of nested loop (for-in-for or a while-in-a-for etc.) in method blocks.
- Nested Selection: This feature collects the number of nested selection statements (if-in-if or a switch-in-a-if etc.) in method blocks.
### A.4 Results of combinations of pair instances vectors

#### Table A.4: Results of distance and multiplicative combinations of pair instances vectors

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Features</th>
<th>Type of Clone</th>
<th>Precision</th>
<th>Recall</th>
<th>F-Measure</th>
<th>Precision</th>
<th>Recall</th>
<th>F-Measure</th>
<th>Precision</th>
<th>Recall</th>
<th>F-Measure</th>
<th>Precision</th>
<th>Recall</th>
<th>F-Measure</th>
</tr>
</thead>
<tbody>
<tr>
<td>Random Forest</td>
<td>Traditional Features (14)</td>
<td>Type1/2</td>
<td>89.2%</td>
<td>100%</td>
<td>94.9%</td>
<td>89.4%</td>
<td>100%</td>
<td>94.7%</td>
<td>89.4%</td>
<td>100%</td>
<td>94.7%</td>
<td>89.4%</td>
<td>100%</td>
<td>94.7%</td>
</tr>
<tr>
<td></td>
<td>VST3</td>
<td>46.6%</td>
<td>97.5%</td>
<td>91.1%</td>
<td>94.7%</td>
<td>91.2%</td>
<td>97.5%</td>
<td>94.7%</td>
<td>91.2%</td>
<td>97.5%</td>
<td>94.7%</td>
<td>91.2%</td>
<td>97.5%</td>
<td>94.7%</td>
</tr>
<tr>
<td></td>
<td>ST3</td>
<td>81.5%</td>
<td>97.5%</td>
<td>94.7%</td>
<td>92.1%</td>
<td>95.7%</td>
<td>98.1%</td>
<td>97.5%</td>
<td>95.7%</td>
<td>98.1%</td>
<td>97.5%</td>
<td>95.7%</td>
<td>98.1%</td>
<td>97.5%</td>
</tr>
<tr>
<td></td>
<td>MT3</td>
<td>78.2%</td>
<td>97.5%</td>
<td>94.7%</td>
<td>92.1%</td>
<td>95.7%</td>
<td>98.1%</td>
<td>97.5%</td>
<td>95.7%</td>
<td>98.1%</td>
<td>97.5%</td>
<td>95.7%</td>
<td>98.1%</td>
<td>97.5%</td>
</tr>
<tr>
<td></td>
<td>VST3/4</td>
<td>89.2%</td>
<td>100%</td>
<td>94.9%</td>
<td>92.1%</td>
<td>95.7%</td>
<td>98.1%</td>
<td>97.5%</td>
<td>95.7%</td>
<td>98.1%</td>
<td>97.5%</td>
<td>95.7%</td>
<td>98.1%</td>
<td>97.5%</td>
</tr>
<tr>
<td></td>
<td>False</td>
<td>88.8%</td>
<td>98%</td>
<td>97.5%</td>
<td>95.2%</td>
<td>97.8%</td>
<td>98%</td>
<td>97.5%</td>
<td>95.2%</td>
<td>97.8%</td>
<td>98%</td>
<td>97.5%</td>
<td>95.2%</td>
<td>97.8%</td>
</tr>
<tr>
<td></td>
<td>Semantic Features (43)</td>
<td>Type1/2</td>
<td>55.7%</td>
<td>100%</td>
<td>93.2%</td>
<td>96.7%</td>
<td>100%</td>
<td>93.2%</td>
<td>96.7%</td>
<td>100%</td>
<td>93.2%</td>
<td>96.7%</td>
<td>100%</td>
<td>93.2%</td>
</tr>
<tr>
<td></td>
<td>VST3</td>
<td>48.4%</td>
<td>86%</td>
<td>91%</td>
<td>92.1%</td>
<td>95.7%</td>
<td>98%</td>
<td>91%</td>
<td>92.1%</td>
<td>95.7%</td>
<td>98%</td>
<td>91%</td>
<td>92.1%</td>
<td>95.7%</td>
</tr>
<tr>
<td></td>
<td>ST3</td>
<td>82.8%</td>
<td>97.5%</td>
<td>94.7%</td>
<td>92.1%</td>
<td>95.7%</td>
<td>98.1%</td>
<td>97.5%</td>
<td>95.7%</td>
<td>98.1%</td>
<td>97.5%</td>
<td>95.7%</td>
<td>98.1%</td>
<td>97.5%</td>
</tr>
<tr>
<td></td>
<td>MT3</td>
<td>76.8%</td>
<td>97.5%</td>
<td>94.7%</td>
<td>92.1%</td>
<td>95.7%</td>
<td>98.1%</td>
<td>97.5%</td>
<td>95.7%</td>
<td>98.1%</td>
<td>97.5%</td>
<td>95.7%</td>
<td>98.1%</td>
<td>97.5%</td>
</tr>
<tr>
<td></td>
<td>VST3/4</td>
<td>89.4%</td>
<td>98%</td>
<td>97.5%</td>
<td>95.2%</td>
<td>97.8%</td>
<td>98%</td>
<td>97.5%</td>
<td>95.2%</td>
<td>97.8%</td>
<td>98%</td>
<td>97.5%</td>
<td>95.2%</td>
<td>97.8%</td>
</tr>
<tr>
<td></td>
<td>False</td>
<td>89.4%</td>
<td>98%</td>
<td>97.5%</td>
<td>95.2%</td>
<td>97.8%</td>
<td>98%</td>
<td>97.5%</td>
<td>95.2%</td>
<td>97.8%</td>
<td>98%</td>
<td>97.5%</td>
<td>95.2%</td>
<td>97.8%</td>
</tr>
<tr>
<td></td>
<td>Syntactic + Semantic Features (86)</td>
<td>Type1/2</td>
<td>55.7%</td>
<td>100%</td>
<td>93.2%</td>
<td>96.7%</td>
<td>100%</td>
<td>93.2%</td>
<td>96.7%</td>
<td>100%</td>
<td>93.2%</td>
<td>96.7%</td>
<td>100%</td>
<td>93.2%</td>
</tr>
<tr>
<td></td>
<td>VST3</td>
<td>48.4%</td>
<td>86%</td>
<td>91%</td>
<td>92.1%</td>
<td>95.7%</td>
<td>98%</td>
<td>91%</td>
<td>92.1%</td>
<td>95.7%</td>
<td>98%</td>
<td>91%</td>
<td>92.1%</td>
<td>95.7%</td>
</tr>
<tr>
<td></td>
<td>ST3</td>
<td>82.8%</td>
<td>97.5%</td>
<td>94.7%</td>
<td>92.1%</td>
<td>95.7%</td>
<td>98.1%</td>
<td>97.5%</td>
<td>95.7%</td>
<td>98.1%</td>
<td>97.5%</td>
<td>95.7%</td>
<td>98.1%</td>
<td>97.5%</td>
</tr>
<tr>
<td></td>
<td>MT3</td>
<td>76.8%</td>
<td>97.5%</td>
<td>94.7%</td>
<td>92.1%</td>
<td>95.7%</td>
<td>98.1%</td>
<td>97.5%</td>
<td>95.7%</td>
<td>98.1%</td>
<td>97.5%</td>
<td>95.7%</td>
<td>98.1%</td>
<td>97.5%</td>
</tr>
<tr>
<td></td>
<td>VST3/4</td>
<td>89.4%</td>
<td>98%</td>
<td>97.5%</td>
<td>95.2%</td>
<td>97.8%</td>
<td>98%</td>
<td>97.5%</td>
<td>95.2%</td>
<td>97.8%</td>
<td>98%</td>
<td>97.5%</td>
<td>95.2%</td>
<td>97.8%</td>
</tr>
<tr>
<td></td>
<td>False</td>
<td>89.4%</td>
<td>98%</td>
<td>97.5%</td>
<td>95.2%</td>
<td>97.8%</td>
<td>98%</td>
<td>97.5%</td>
<td>95.2%</td>
<td>97.8%</td>
<td>98%</td>
<td>97.5%</td>
<td>95.2%</td>
<td>97.8%</td>
</tr>
</tbody>
</table>

*Note: The table contains precision, recall, and F-measure values for different conditions and algorithms.*
### A.5 Results of BigCloneBench Recall, Precision and F-Measure Measurements

Table A.5: BigCloneBench Recall, Precision and F-Measure Measurements. Existing detectors results are obtained from Sajnani et al. [96].

<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></td>
<td>94.8</td>
</tr>
<tr>
<td></td>
<td>VST3</td>
<td>93</td>
<td></td>
<td>92</td>
</tr>
<tr>
<td></td>
<td>ST3</td>
<td>61</td>
<td></td>
<td>73</td>
</tr>
<tr>
<td></td>
<td>MT3</td>
<td>5</td>
<td></td>
<td>9</td>
</tr>
<tr>
<td></td>
<td>WT3/4</td>
<td>0</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>91 (as reported)</td>
<td></td>
</tr>
<tr>
<td>CCFinder</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>61</td>
</tr>
<tr>
<td></td>
<td>ST3</td>
<td>15</td>
<td></td>
<td>24</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></td>
<td></td>
<td></td>
<td>60 – 72 (as reported)</td>
<td></td>
</tr>
<tr>
<td>Deckard</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>47</td>
</tr>
<tr>
<td></td>
<td>ST3</td>
<td>31</td>
<td></td>
<td>21</td>
</tr>
<tr>
<td></td>
<td>MT3</td>
<td>12</td>
<td></td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>WT3/4</td>
<td>1</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>93 (as reported)</td>
<td></td>
</tr>
<tr>
<td>iClones</td>
<td>Type1/2</td>
<td>91</td>
<td></td>
<td>98</td>
</tr>
<tr>
<td></td>
<td>VST3</td>
<td>82</td>
<td></td>
<td>89 – 98</td>
</tr>
<tr>
<td></td>
<td>ST3</td>
<td>24</td>
<td></td>
<td>87 – 95</td>
</tr>
<tr>
<td></td>
<td>MT3</td>
<td>0</td>
<td></td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>WT3/4</td>
<td>0</td>
<td></td>
<td>(Unreported)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>(Unreported)</td>
<td></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>(Unreported)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>80 – 96 (as reported)</td>
<td></td>
</tr>
<tr>
<td>Our Method Using Distance Fusion</td>
<td>Type1/2</td>
<td>100</td>
<td>90.7</td>
<td>95.1</td>
</tr>
<tr>
<td></td>
<td>VST3</td>
<td>64</td>
<td>94.2</td>
<td>76.2</td>
</tr>
<tr>
<td></td>
<td>ST3</td>
<td>94.9</td>
<td>95.7</td>
<td>95.3</td>
</tr>
<tr>
<td></td>
<td>MT3</td>
<td>94</td>
<td>91.7</td>
<td>92.8</td>
</tr>
<tr>
<td></td>
<td>WT3/4</td>
<td>94.7</td>
<td>95.4</td>
<td>95</td>
</tr>
<tr>
<td>Our Method Using Multiplicative</td>
<td>Type1/2</td>
<td>92.1</td>
<td>96.1</td>
<td>94.1</td>
</tr>
<tr>
<td></td>
<td>VST3</td>
<td>93.6</td>
<td>91.8</td>
<td>92.7</td>
</tr>
<tr>
<td></td>
<td>ST3</td>
<td>95.4</td>
<td>94</td>
<td>94.7</td>
</tr>
<tr>
<td></td>
<td>MT3</td>
<td>92.8</td>
<td>91.9</td>
<td>92.3</td>
</tr>
<tr>
<td></td>
<td>WT3/4</td>
<td>94.3</td>
<td>95</td>
<td>94.6</td>
</tr>
<tr>
<td>Our Method Using Linear Fusion</td>
<td>Type1/2</td>
<td>92.1</td>
<td>95.7</td>
<td>93.9</td>
</tr>
<tr>
<td></td>
<td>VST3</td>
<td>93.2</td>
<td>91.9</td>
<td>92.6</td>
</tr>
<tr>
<td></td>
<td>ST3</td>
<td>97.6</td>
<td>95.8</td>
<td>96.7</td>
</tr>
<tr>
<td></td>
<td>MT3</td>
<td>95.8</td>
<td>93.7</td>
<td>94.7</td>
</tr>
<tr>
<td></td>
<td>WT3/4</td>
<td>95.4</td>
<td>97.1</td>
<td>96.2</td>
</tr>
</tbody>
</table>
### Appendix B

### FEATURES OF CHAPTER 6

#### B.1 More AST Features

Table B.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 Declaration</td>
<td>Counts &quot;[ Javadoc ] ExtendedModifier Type Identifier ( ) [ default Expression ] ;&quot; in method blocks.</td>
</tr>
<tr>
<td>No. of Anonymous Class Declaration1</td>
<td></td>
<td>Counts &quot;ClassBodyDeclaration&quot; in method blocks.</td>
</tr>
<tr>
<td>No. of Annotation TypeDeclaration</td>
<td></td>
<td>Counts &quot;[ Javadoc ] ExtendedModifier @ interface Identifier AnnotationTypeBodyDeclaration &quot; ;&quot; in method blocks.</td>
</tr>
<tr>
<td>No. of String Literal1</td>
<td></td>
<td>Counts &quot;public class StringLiteral extends Expression&quot; in method blocks.</td>
</tr>
<tr>
<td>No. of Package Declaration</td>
<td></td>
<td>Counts &quot;[ Javadoc ] Annotation package Name ;&quot; in method blocks.</td>
</tr>
<tr>
<td>No. of Number Literal</td>
<td></td>
<td>Counts &quot;public class NumberLiteral extends Expression&quot; in method blocks.</td>
</tr>
<tr>
<td>No. of Name Qualified Type</td>
<td></td>
<td>Counts &quot;Name . Annotation SimpleName&quot; in method blocks.</td>
</tr>
<tr>
<td>No. of Modifier</td>
<td></td>
<td>Counts &quot;public final class Modifier extends ASTNode implements IExtendedModifier&quot; in method blocks.</td>
</tr>
<tr>
<td>No. of Member Ref</td>
<td></td>
<td>Counts &quot;[ Name ] Identifier &quot; in method blocks.</td>
</tr>
<tr>
<td>No. of Marker Annotation</td>
<td></td>
<td>Counts &quot;@ TypeName&quot; in method blocks.</td>
</tr>
<tr>
<td>No. of Javadoc</td>
<td></td>
<td>Counts &quot;/** TagElement */&quot; in method blocks.</td>
</tr>
<tr>
<td>No. of Initializer</td>
<td></td>
<td>Counts &quot;[ static ] Block&quot; in method blocks.</td>
</tr>
<tr>
<td>No. of Enum Constant Declaration1</td>
<td></td>
<td>Counts &quot;[ Javadoc ] ExtendedModifier Identifier { [ Expression , Expression ] } &quot; in method blocks.</td>
</tr>
<tr>
<td>No. of Empty Statement</td>
<td></td>
<td>Counts &quot;;&quot; in method blocks.</td>
</tr>
<tr>
<td>No. of Dimension</td>
<td></td>
<td>Counts &quot;Annotation [ ]&quot; in method blocks.</td>
</tr>
<tr>
<td>No. of Compilation Unit</td>
<td></td>
<td>Counts &quot;[ PackageDeclaration ] ImportDeclaration TypeDeclaration — EnumDeclaration — AnnotationTypeDeclaration ;&quot; in method blocks.</td>
</tr>
<tr>
<td>No. of Character Literal</td>
<td></td>
<td>Counts &quot;public class CharacterLiteral extends Expression&quot; in method blocks.</td>
</tr>
<tr>
<td>No. of Super Constructor Invocation</td>
<td></td>
<td>Counts &quot;[ Expression . [ &lt;Type , Type &gt;] super ( [ Expression , Expression ] ) ;&quot; in method blocks.</td>
</tr>
<tr>
<td>No. of Single Member Annotation</td>
<td></td>
<td>Counts &quot;@ TypeName ( Expression )&quot; in method blocks.</td>
</tr>
<tr>
<td>No. of Normal Annotation</td>
<td></td>
<td>Counts &quot;@ TypeName ( [ MemberValuePair , MemberValuePair ] ) ;&quot; in method blocks.</td>
</tr>
</tbody>
</table>
## B.2 BDG Features

Table B.2: Some of 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 ConditionalBranchAload_0</td>
<td>Counts <code>ConditionalBranch → Aload\_0</code> in method blocks.</td>
</tr>
<tr>
<td></td>
<td>No. of CondZeroAload_0</td>
<td>Counts <code>CondZero → Aload</code> in method blocks.</td>
</tr>
<tr>
<td></td>
<td>No. of CondCompareAload_0</td>
<td>Counts <code>CondCompare → Aload\_0</code> in method blocks.</td>
</tr>
<tr>
<td></td>
<td>No. of ConditionalBranchAload_1</td>
<td>Counts <code>ConditionalBranch → Aload\_1</code> in method blocks.</td>
</tr>
<tr>
<td></td>
<td>No. of CondZeroAload_1</td>
<td>Counts <code>CondZero → Aload\_1</code> in method blocks.</td>
</tr>
<tr>
<td></td>
<td>No. of CondCompareAload_1</td>
<td>Counts <code>CondCompare → Aload\_1</code> in method blocks.</td>
</tr>
<tr>
<td></td>
<td>No. of ConditionalBranchIload_0</td>
<td>Counts <code>ConditionalBranch → Iload\_0</code> in method blocks.</td>
</tr>
<tr>
<td></td>
<td>No. of CondZeroIload_0</td>
<td>Counts <code>CondZero → Iload</code> in method blocks.</td>
</tr>
<tr>
<td></td>
<td>No. of CondCompareIload_0</td>
<td>Counts <code>CondCompare → Iload\_0</code> in method blocks.</td>
</tr>
<tr>
<td></td>
<td>No. of ConditionalBranchIstore_3</td>
<td>Counts <code>ConditionalBranch → Istore\_3</code> in method blocks.</td>
</tr>
<tr>
<td></td>
<td>No. of CondZeroIstore_3</td>
<td>Counts <code>CondZero → Istore\_3</code> in method blocks.</td>
</tr>
<tr>
<td></td>
<td>No. of CondCompareIstore_3</td>
<td>Counts <code>CondCompare → Istore\_3</code> in method blocks.</td>
</tr>
<tr>
<td></td>
<td>No. of ConditionalBranchIstore_2</td>
<td>Counts <code>ConditionalBranch → Istore\_2</code> in method blocks.</td>
</tr>
<tr>
<td></td>
<td>No. of CondZeroIstore_2</td>
<td>Counts <code>CondZero → Istore\_2</code> in method blocks.</td>
</tr>
<tr>
<td></td>
<td>No. of CondCompareIstore_2</td>
<td>Counts <code>CondCompare → Istore\_2</code> in method blocks.</td>
</tr>
<tr>
<td></td>
<td>No. of ConditionalBranchIstore_1</td>
<td>Counts <code>ConditionalBranch → Istore\_1</code> in method blocks.</td>
</tr>
<tr>
<td></td>
<td>No. of CondZeroIstore_1</td>
<td>Counts <code>CondZero → Istore\_1</code> in method blocks.</td>
</tr>
<tr>
<td></td>
<td>No. of CondCompareIstore_1</td>
<td>Counts <code>CondCompare → Istore\_1</code> in method blocks.</td>
</tr>
<tr>
<td></td>
<td>No. of ConditionalBranchIstore_0</td>
<td>Counts <code>ConditionalBranch → Istore\_0</code> in method blocks.</td>
</tr>
<tr>
<td></td>
<td>No. of CondZeroIstore_0</td>
<td>Counts <code>CondZero → Istore\_0</code> in method blocks.</td>
</tr>
<tr>
<td></td>
<td>No. of CondCompareIstore_0</td>
<td>Counts <code>CondCompare → Istore\_0</code> in method blocks.</td>
</tr>
<tr>
<td></td>
<td>No. of ConditionalBranchIinc</td>
<td>Counts <code>ConditionalBranch → Iinc</code> in method blocks.</td>
</tr>
<tr>
<td></td>
<td>No. of CondZeroIinc</td>
<td>Counts <code>CondZero → Iinc</code> in method blocks.</td>
</tr>
<tr>
<td></td>
<td>No. of CondCompareIinc</td>
<td>Counts <code>CondCompare → Iinc</code> in method blocks.</td>
</tr>
<tr>
<td></td>
<td>No. of ConditionalBranchNew</td>
<td>Counts <code>ConditionalBranch → New</code> in method blocks.</td>
</tr>
<tr>
<td></td>
<td>No. of CondZeroNew</td>
<td>Counts <code>CondZero → New</code> in method blocks.</td>
</tr>
<tr>
<td></td>
<td>No. of CondCompareNew</td>
<td>Counts <code>CondCompare → New</code> in method blocks.</td>
</tr>
</tbody>
</table>