UNIVERSITY OF CALIFORNIA,
IRVINE

Towards Accurate and Scalable Clone Detection using Software Metrics

DISSERTATION

submitted in partial satisfaction of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

in Software Engineering

by

Vaibhav Pratap Singh Saini

2018

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

There are many people who have my gratitude for providing me with unfailing support and continuous encouragement throughout my years of study, research and writing this thesis. In particular I would like to thank the dissertation committee members, my lab mates, the funding entities and my friends and family.

My advisor. I am indebted to my thesis advisor, Prof. Cristina Lopes to have given me tremendous academic support and for the wonderful opportunities contributing to my research. Crista let me explore the problem space myself and was always there to hear me out and brainstorm the ideas with me. She not only gave me insights to go about with my problem statements but also guided me in the right direction to find the best possible solutions. Her guidance and advice on both research as well as my career has been priceless.

The dissertation committee members. Besides my advisor, I am very grateful to my committee members, Prof. Sam Malek and Prof. James A. Jones for their insightful comments, crucial remarks and invaluable advice.

My labmates. I would further like to thank and express my deep appreciation towards my labmates, especially Hitesh Sajnani and Farima Farahani Farmahini. Hitesh was always there to motivate me and was next to being my mentor right from the time I joined UCI while Farima helped me immensely throughout my research work for Oreo and InspectorClone. I am also thankful to Pedro Martins and Di Yang for their thought provoking suggestions and collaboration as well as to Arthur Valadares, Rohan Achar, Thomas Debeauvais, Eugenia Gabrielova and Wen Shen for the collegiality that each of them offered to me over the years and for making Mondego lab a fun place to work.

My friends and family. I am profoundly thankful for the continuous support of my parents, especially my mother who constantly encouraged me to work hard and give my best and for the unending sacrifices she made throughout her life to get me to achieve what I have today. Despite the distance, she made sure to be with me in spirit and instilled in me the confidence which was much needed in times of lockdowns which I faced with my problem statements during the tenure of my research. I cannot thank my father enough for all the positivity he induced in the stressful times, encouraging me to make time for fun and happiness in all situations, which helped me sustain the hours of hardship. I also would like to thank my brother, who I could always fall back on, for hearing out my ideas over a phone call in the middle of the night, for giving his valuable inputs and for supporting my decisions. I am also deeply grateful to my friends Aditya, Gautam and Garima for keeping me in high spirits during tough times and to have made themselves available for trips and excursions for me to have a good work life balance.

Lastly, I owe an immense debt of gratitude to my loving wife Sneha, for the innumerable sacrifices she made in shouldering far more than her fair share of household burdens while I pursued my research. She took a great deal of responsibility in her stride and supported me in every manner possible as I worked through late nights for deadlines and submissions.

She was patient enough to hear out my ideas, read and learn about them and also critique despite her not being formally trained in computer science.

The funding entities. This work was partially supported by a grant from the National Science Foundation No.1218228, by the DARPA MUSE program, and by the Department of Informatics at University of California Irvine.

CURRICULUM VITAE

Vaibhav Pratap Singh Saini

VAIBHAV SAINI
linkedin: https://www.linkedin.com/in/sainivaibhav
email: vaibps17@gmail.com
phone: +1 (949) 887-7143

SUMMARY

My area of research has been in the field of Software Engineering with an emphasis on creating tools and knowledge for software engineers and researchers. I use information retrieval, statistical, and machine learning techniques to analyze software systems.

I have developed state of the art source code clone detectors, namely Oreo, SourcererCC, and SourcererCC-I.

EDUCATION

Ph.D. Software Engineering— GPA 3.97/4.00 *Sept. 2012 to Dec. 2018*
University of California, Irvine, CA
Dissertation Title: Towards the accurate and large scale detection of complex code clones using software metrics.

M.S. Software Engineering— GPA 3.94/4.00 *Sept. 2012 to Dec. 2018*
University of California, Irvine, CA

B.Tech. Information and Communication Technology *Sept. 2004 to May. 2008*
Dheerubhai Ambani Institute of Information and Communication Technology, Gandhinagar, Gujarat, India

Selected Awards and Honors

Best Paper Award: V. Saini, H. Sajnani, and C. Lopes. Comparing quality metrics for cloned and non cloned java methods: A large scale empirical study. In 2016 IEEE International Conference on Software Maintenance and Evolution (ICSME), Oct 2016.

Distinguished Artifact Award: C. V. Lopes, P. Maj, P. Martins, V. Saini, D. Yang, J. Zitny, H. Sajnani, and J. Vitek. Déjàvu: A Map of Code Duplicates on Github. Proc. ACM Program. Lang., 1(OOPSLA), Oct. 2017.

ESEC/FSE 2018 Distinguished Paper Award: Vaibhav Saini, Farima Farmahinifarahani, Yadong Lu, Pierre Baldi, and Cristina V. Lopes. 2018. Oreo: Detection of Clones in the Twilight Zone. In Proceedings of the 26th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '18), November 4–9, 2018, Lake Buena Vista, FL, USA. ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3236024.3236026

Career History

Graduate Research Assistant, UC Irvine, *Advisor: Prof. Cristina Lopes Winter 2012 - December 2018*

- Machine Learning based approaches to software architecture recovery
- Scalable code clone detection techniques
- Mining software repositories to gain insights in the context of software evolution

Researcher, VSTS Intern, MSR, Redmond, *Mentor: Stanislaw Swierc Summer 2016*

- Integrating Stack-Trace Search into Visual Service Team Services, an engineering platform.
- Conducting experiments to evaluating different full text search options for Stack-Trace Search.

- Communicating with distributed teams.

Intern, SRCH2, Irvine, *Mentor: Prof. Chen Li, Founder* *Summer 2013*

- SRCH2 is a start-up in enterprise search solutions. I worked on building a query parser to parse Solr like queries.

Senior Software Developer, Kuliza Technologies, Bangalore *Sept. 2008 - Aug. 2012*

- GTeam: (2012). Multi-threaded J2SE desktop client to track time, take automatic screenshots, take automatic webcam image, keyboard hits and mouse clicks.
- TxtWeb(2011-2012). A platform for creating and publishing SMS based apps to provide information from the Internet.(PHP, Javascript, Joomla.)
- Modular Framework for Java:(2010). Open source initiative to support multiple versions of a module in a runtime environment.

Publications

- V. Saini, F. Farmahinifarahani, Y. Lu, D. Yang, P. Martins, H. Sajnani,P. Baldi, and C. Lopes, "Towards Automating Precision Studies of Clone Detectors," Submitted in ICSE 2019.

- V. Saini, F. Farmahinifarahani, Y. Lu, P. Baldi, and C. Lopes, "Oreo: Detection of clones in the Twilight Zone," in Proceedings of the 2018 26th ACM SIGSOFT International Symposium on Foundations of Software Engineering (To Appear), FSE 2018, (New York, NY, USA), ACM, 2018. https://arxiv.org/abs/1806.05837.

- C. V. Lopes, P. Maj, P. Martins, V. Saini, D. Yang, J. Zitny, H. Sajnani, and J. Vitek. Déjàvu: A Map of Code Duplicates on Github. Proc. ACM Program. Lang., 1(OOPSLA):84:1–84:28, Oct. 2017.

- D. Yang, P. Martins, V. Saini, and C. Lopes. Stack Overflow in Github: Any Snippets There? In 2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR), pages 280–290, May 2017.

- V. Saini, H. Sajnani, and C. Lopes. Cloned and non-cloned Java Methods: A Comparative Study. Empirical Software Engineering, Dec 2017.
- V. Saini, H. Sajnani, and C. Lopes. Comparing Quality Metrics for Cloned and Non Cloned Java Methods: A Large Scale Empirical Study. In 2016 IEEE International Conference on Software Maintenance and Evolution (ICSME), pages 256–266, Oct 2016.
- V Saini, H Sajnani, J Kim, C Lopes. SourcererCC and SourcererCC-I: Tools to Detect Clones in Batch Mode and During Software Development. In Proceedings of the 38th International Conference on Software Engineering (ICSE), 2016
- H. Sajnani, V. Saini, J. Svajlenko, C. K. Roy, and C. V. Lopes. Sourcerercc: Scaling Code Clone Detection to Big-code. In Proceedings of the 38th International Conference on Software Engineering, ICSE '16, pages 1157–1168, New York, NY, USA, 2016. ACM.
- H. Sajnani, V. Saini, and C. Lopes. A Parallel and Efficient Approach to Large Scale Clone Detection. Journal of Software: Evolution and Process, 27(6):402–429, 2015.ware: Evolution and Process, 2015 [19]
- H. Sajnani, V. Saini, J. Ossher, and C. V. Lopes. Is Popularity a Measure of Quality? An Analysis of Maven Components. In 2014 IEEE International Conference on Software Maintenance and Evolution, pages 231–240, Sept 2014.
- H. Sajnani, V. Saini, and C. V. Lopes. A Comparative Study of Bug Patterns in Java Cloned and Non Cloned Code. In 2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation, pages 21–30, Sept 2014.
- V. Saini, H. Sajnani, J. Ossher, and C. V. Lopes. A Dataset for Maven Artifacts and Bug Patterns Found in Them. In Proceedings of the 11th Working Conference on Mining Software Repositories, MSR 2014, pages 416–419, New York, NY, USA, 2014. ACM.

Public Research Talks & Presentations

- **Presenter**, Oreo: Detection of Clones in the Twilight Zone. *In Proceedings of the 26th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering. (ESEC/FSE 2018)*, 2018.
- **Presenter**, OreoCF: Towards Accurate and Scalable Detection of Semantic Clones. Poster Track *In Proceedings of the 40th International Conference on Software Engineering. ACM.* 2018.

- **Presenter**, SourcererCC and SourcererCC-I: tools to detect clones in batch mode and during software development, *In Proceedings of the 38th International Conference on Software Engineering. ACM*. 2016.

- **Presenter**, A Comparative Study of Bug Patterns in Java Cloned and Non-cloned Code, *Proceedings of Working Conference on Source Code Analysis and Manipulation*. 2014.

- **Student Volunteer**, Program Committee meeting held for OOPSLA/SPLASH 2013 in Irvine, CA, USA

- **Presenter**, Creating a scalable contest app on Facebook Platform, *Facebook Developer Garage, Bangalore*. 2011.

ABSTRACT OF THE DISSERTATION

Towards Accurate and Scalable Clone Detection using Software Metrics

By

Vaibhav Pratap Singh Saini

Doctor of Philosophy in Software Engineering

University of California, Irvine, 2018

Professor Cristina Lopes, Chair

Code clone detection tools find exact or similar pieces of code, known as code clones. Code clones are categorized into four types of increasing difficulty of detection, ranging from purely textual (Type I) to purely semantic (Type IV). Most clone detectors reported in the literature, work well up to Type III, which accounts for syntactic differences. In between Type III and Type IV, however, there lies a spectrum of clones that, although still exhibiting some syntactic similarities, are extremely hard to detect – the Twilight Zone. Besides correctness, scalability has become a must-have requirement for modern clone detection tools. The increase in amount of source code in web-hosted open source repository services has presented opportunities to improve the state of the art in various modern use cases of clone detection such as detecting similar mobile applications, license violation detection, mining library candidates, code repair, and code search among others. Though these opportunities are exciting, scaling such vast corpora poses critical challenge.

Over the years, many clone detection techniques and tools have been developed. One class of these techniques is based on software metrics. Metrics based clone detection has potential to identify clones in the Twilight Zone. For various reasons, however, metrics-based techniques are hard to scale to large datasets. My work highlights issues which prohibit metric based clone detection techniques to scale large datasets while maintaining high levels of correctness.

The identification of these issues allowed me to rethink how metrics could be used for clone detection.

This dissertation starts by presenting an empirical study using software metrics to understand if metrics can be used to identify differences in cloned and non-cloned code. The study is followed by another large scale study to explore the extent of cloning in GitHub. Here, the dissertation highlights scalability challenges in clone detection and how they were addressed. The above two studies provided a strong base to use software metrics for clone detection in a scalable manner. To this end, the dissertation presents Oreo, a novel approach capable of detecting harder-to-detect clones in the Twilight Zone. Oreo is built using a combination of machine learning, information retrieval, and software metrics. This dissertation evaluates the recall of Oreo on BigCloneBench, a benchmark of real world code clones. In experiments to compare the detection performance of Oreo with other five state of the art clone detectors, we found that Oreo has both high recall and precision. More importantly, it pushes the boundary in detection of clones with moderate to weak syntactic similarity, in a scalable manner. Further, to address the issues identified in precision evaluations, the dissertation presents InspectorClone, a semi automated approach to facilitate precision studies of clone detection tools. InspectorClone makes use of some of the concepts introduced in the design of Oreo to automatically resolve different types of clone pairs. Experiments demonstrate that InspectorClone has a very high precision and it significantly reduces the number of clone pairs that need human validation during precision experiments. Moreover, InspectorClone aggregates the individual effort of multiple teams into a single evolving dataset of labeled clone pairs, creating an important asset for software clone research. Finally, the dissertation concludes with a discussion on the lessons learned during the design and development of Oreo and lists down a few areas for the future work in code clone detection.

Chapter 1

Introduction

Source code clone detection is the task of finding similar software pieces, according to a certain concept of similarity. These pieces can be statements, blocks of code, functions, classes, or even complete source files, and their similarity can be syntactic, semantic or both. Over the past 20 years, clone detection has been the focus of increasing attention, with many clone detectors having been proposed and implemented (see [142] for a recent survey on this topic). These clone detection techniques and tools differ from each other depending on the goals and granularity of the detection. Modern use cases of clone detection [98] require clone detectors to not only be accurate but also scalable. Recently, researchers have proposed and demonstrated techniques that are accurate and can scale to large datasets [137, 152]. These techniques however fall short of detecting complex clones where the syntactic similarity between the code snippets is low. The accurate detection of such complex clones in a scalable manner is still an active area of research.

1.1 Terminology

In this section, I elaborate the following well-accepted terms and definitions that are pivotal in my dissertation [18, 19, 68, 126, 148].

1.1.1 Code Clone Terms

Code Fragment or Block: A continuous segment of source code specified by the triple (l,s,e), including the source file, l, the start line of the fragment, s, and the end line, e.

Clone Pair: A pair of code fragments that are similar, specified by the triple (f1, f2, ϕ), including the similar code fragments f1 and f2, and their clone type ϕ.

Near-miss Clone: Code fragments that have minor to significant editing differences between them.

1.1.2 Code Clone Types

In the literature, four commonly accepted clone types can be found:

- Type I (textual similarity): Identical code fragments, except for differences in white-space, layout and comments.
- Type II (lexical, or token-based, similarity): Identical code fragments, except for differences in identifier names and literal values, in addition to Type-1 clone differences.
- Type III (syntactic similarity): Syntactically similar code fragments that differ at the statement level. The fragments have statements added, modified and/or removed with respect to each other, in addition to Type-1 and Type-2 clone differences

- Type IV (semantic similarity): Syntactically dissimilar code fragments that implement the same functionality.

Clone detectors use a variety of signals from the code (text, tokens, syntactic features, program dependency graphs, etc.) and tend to aim for detecting specific types of clones, usually up to Type III. Very few of them attempt at detecting pure Type-4 clones, since it requires analysis of the actual behavior – a hard problem, in general.

1.1.3 The Twilight Zone

Starting at Type III and onwards lies a spectrum of clones that, although still within the reach of automatic clone detection, are increasingly hard to detect. Reflecting the vastness of this spectrum, the popular clone benchmark BigCloneBench [150] includes subcategories between Type III and Type IV, namely Very Strongly Type III, Strongly Type III, Moderately Type III, and Weakly Type III, which merges with Type IV.

In order to illustrate the spectrum of clone detection, and its challenges, Listing 1.1 shows one example method followed by several clones of it, from Type-2 to Type-4. The original method takes two numbers and returns a comma-separated sequence of integers in between the two numbers, as a string. The Type-2 clone (starting in line #13) is syntactically identical, and differs only in the identifiers used (e.g. `begin` instead of `start`). It is very easy for clone detectors to identify this type of clones. The very strong Type-3 clone (starting in line #25) has some lexical as well as syntactic differences, namely the use of a for-loop instead of a while-loop. Altough harder than Type-2, this subcategory of Type-3 is still relatively easy to detect. The moderate Type-3 clone (starting in line #35) differs even more from the original method: the name of the method is different (`seq` vs. `sequence`), the comma is placed in its own local variable, and the type String is used instead of StringBuilder. This subcategory of Type-3 clones is much harder to detect than the previous ones. The weak Type-3 clone

Listing 1.1: Sequence Between Two Numbers

```
// Original method
String sequence(int start, int stop) {
    StringBuilder builder = new StringBuilder();
    int i = start;
    while (i <= stop) {
        if (i > start) builder.append(',');
        builder.append(i);
        i++;
    }
    return builder.toString();
}

// Type-2 clone
String sequence(int begin, int end) {
    StringBuilder builder = new StringBuilder();
    int n = begin;
    while (n <= end) {
        if (n > begin) builder.append(',');
        builder.append(n);
        n++;
    }
    return builder.toString();
}

// Very strongly Type-3 clone
String sequence(short start, short stop) {
    StringBuilder builder = new StringBuilder();
    for (short i = start; i <= stop; i++) {
        if (i > start) builder.append(',');
        builder.append(i);
    }
    return builder.toString();
}

// Moderately Type-3 clone
String seq(int start, int stop){
    String sep = ",";
    String result = Integer.toString(start);
    for (int i = start + 1; ; i++) {
        if (i > stop) break;
        result = String.join(sep, result, Integer.toString(i));
    }
    return result;
}

// Weakly Type-3 clone
String seq(int begin, int end, String sep){
    String result = Integer.toString(begin);
    for (int n = begin + 1; ;n++) {
        if (end < n)  break;
        result = String.join(sep, result, Integer.toString(n));
    }
    return result;
}

// Type-4 clone
String range(short n, short m){
    if (n == m)
    return Short.toString(n);
    return Short.toString(n)+ "," + range(n+1, m);
}
```

(starting in line#46) differs from the original method by a combination of lexical, syntactic and semantic changes: String vs. StringBuilder, a conditional whose logic has changed (< vs >), and it takes one additional input parameter that allows for different separators. The similarities here are weak and very hard to detect. Finally, the Type-4 clone (starting in line #56) implements similar (but not the exact same) functionality in a completely different manner (through recursion), and it has almost no lexical or syntactic similarities to the original method. Detecting Type-4 clones, in general, requires a deep understanding of the intent of a piece of code, especially because the goal of clone detection is similarity, and not exact equivalence (including for semantics). Clones that are moderately Type-3 and onward fall in the *Twilight Zone* of clone detection.

1.2 Motivation

The advent of web-hosted open source repository services such as GitHub, BitBucket and SourceForge have transformed how source code is shared. Over the last two decades, millions of projects have been shared, building up a massive trove of free software. The availability of such a large corpus of source code has attracted clone researchers to mine it in the hopes of finding patterns of interest. Such large corpora present the opportunity to improve the state of the art in various modern use cases of clone detection. Some of these use cases include studying the extent of cloning within and across code hosting repository platforms [99, 162], studying patterns of clone evaluation [17, 79], conducting empirical studies to understand the characteristics of clones [133], detecting similar mobile applications [28], license violation detection [44, 85], mining library candidates [60], reverse engineering product lines [44, 56], code search [74], and finding the provenance of a component [32]. Though these opportunities are exciting, scaling such vast corpora poses critical challenge making scalability a must-have requirement of modern clone detection tools. This need for scalability in clone detection tools can also be found in literature [78, 84, 99, 137]. Quoting Koschke, "*Detecting license violations of source code requires to compare a suspected system against a very large corpus of source code, for instance, the Debian source distribution. Thus, code clone detection techniques must scale in terms of resources needed.*"

As mentioned earlier, clones that are moderately Type III and onward fall in the *Twilight Zone* of clone detection. Existing studies show that in the software systems, the Type III clones outnumber the clones of other types [129, 146]. This makes the detection of Type III clones even more important. Further, in our experience we have found that there exist more clone pairs in the Twilight Zone than in the simpler Type III subcategories, making it desirable to push the boundaries of clone detection to cover the Twilight Zone.

1.3 Problem Statement

Many tools and techniques have been published to detect source code clones. For example, in 2013, in a systematic literature review, Rattan et al. found at least 70 clone detection tools and techniques [123]. Since then, more tools and techniques, which have improved the state of the art have been built [132, 137, 152]. While some of these modern tools detect clones up to Type III in a scalable manner, they fall short of detecting clones in the Twilight Zone.

Clone detection in general requires comparison among every code fragment to detect all possible clones in a given dataset. This makes clone detection bear a prohibitive $O(n^2)$ time complexity. With the increase in size of datasets, the number of pairs that need to be compared increases quadratically: a candidate explosion problem, posing serious scalability challenges.

A general strategy for scaling clone detection is to eliminate unlikely clone pairs upfront i.e., before making them undergo heavy computational operations. As mentioned earlier, it becomes harder to detect a clone pair as we move from Type I to Type IV because the computations get more expensive. Other than the increase in computation load, the number of pairs that need to be compared also increases drastically.

There exist token based scalable techniques that use aggressive heuristics to filter out pairs which are usually beyond *ST3* category [137, 152]. This enables them to scale large datasets at the cost of poor detection performance in the Twilight Zone.

These scalable token based techniques use bag-of-words model to detect near-miss Type III clones. The code fragments in Type III clones are created by adding, removing, or modifying statements in a duplicated code fragment. The bag-of-tokens model is resilient to such changes because it is agnostic to the relative positions of tokens in code fragments. This model has been shown to detect near-miss clones as long as the code fragments share enough tokens to exceed a given threshold. The similarity in tokens of the clone pairs in the Twilight Zone drops significantly making such token based techniques ineffective in detecting clones in the Twilight Zone.

Metrics based approaches for clone detection are known to work very well if the goal is to find only Type I and Type II clones [81, 105, 120]. This is understandable: given the strict definitions of Type I and Type II, the metrics values of such clone pairs should be mostly the same. For Type III, metrics might look like a good choice too because metrics are resilient to changes in identifiers and literals. However, the use of metrics for clones in the Twilight Zone is not straightforward, because these clones may be syntactically different. As such, the use of metrics requires fine tuning over a large number of configurations between the thresholds of each individual metric leading to an explosion of configuration options. Finding the right balance manually can be hard, for example, is *the number of conditionals* more meaningful than the *number of arguments*? Apart from the configuration explosion problem, metric based techniques have poor precision in Type III category and also suffer from the candidate explosion problem.

Accurate and scalable detection of code clones in the Twilight Zone remains a difficult problem to address.

1.4 Thesis

The above problems motivate the need for clone detection techniques which satisfy the following requirements: (1) Accurate detection of clones in the Twilight Zone; (2) Scalability to large datasets of source code without the necessity of special hardware. To address the above requirements, I propose Oreo, a novel approach to source code clone detection, that not only detects Type I to Type III clones accurately, but is also capable of detecting clones in the Twilight Zone. Oreo is built using a combination of machine learning, information retrieval and software metrics. Oreo uses a novel filtering heuristic to address the problem of candidate explosion. This heuristic selects pairs where the semantic similarity between the code fragments is high. Moreover, to identify the structural similarity between the code fragments of these pairs, Oreo uses a deep neural network model trained using software metrics.Oreo currently supports method level clone detection in Java.

I can now state my thesis as follows: *Oreo improves the state of the art in code clone detection by being the most scalable and the most effective technique known so far to detect clones in the Twilight Zone.*

I compare Oreo's detection performance against the latest versions of five publicly available clone detection tools, namely, SourcererCC [137], NiCad [127], CloneWorks [152], Deckard [64], and CCAligner [156]. These tools are considered state of the art in the literature as well as in recent clone detection benchmarking experiments. They are also examples of modern clone detection tools that support Type III clone detection.

To measure recall and precision, I use a popular benchmark of real clones, Big-CloneBench [148]. To measure scalability, I use IJaDataset-2.0, a dataset of large inter-project software repositories consisting of 25,000 projects containing 3 million files and 250 MLOC.

1.5 Contributions

This dissertation makes the following contributions:

(i) Empirical study to understand if cloned methods are different from non-cloned methods. The study explores the relationship between code clones and 27 software quality metrics on a dataset of 3,562 Java projects.(Chapter 3: Study1)

(ii) Large scale quantitative and qualitative study to explore the extent of cloning in GitHub, a popular source code repository hosting service. We conducted the study for four popular languages, Java, C/C++, Python, and Javascript. In total, the study analyzes 427,807,061 files across 4,494,438 projects. The study posed challenges related to scalability and execution time of SourcererCC, a state of the art clone detector which we used to compute clone detection. This work describes these scalability challenges and how I addressed them. (Chapter 3: Study 2)

(iii) Oreo, a scalable and effective clone detector capable of detecting clones in the Twilight Zone. To design Oreo, I use a subset of software metrics used in the empirical study as referred above in contribution (i). This study addresses scalability and accuracy challenges, the two major issues which software metric based clone detection approaches usually suffer from. To scale to large datasets, Oreo uses the concepts which I developed while addressing scalability issues in SourcererCC. (Chapter 4)

(iv) Experiments to evaluate scalability and effectiveness (recall and precision) against existing state of the art tools. The results of these experiments can be used as benchmarks to compare and evaluate future clone detection tools and techniques. (Chapter 5)

(v) InspectorClone, a web application which facilitates in conducting precision experiments of clone detection tools. InspectorClone significantly reduces the manual effort involved in such precision studies. InspectorClone contributes to the clone research community by making an ongoing dataset of manually tagged clone pairs available publicly for research purposes. (Chapter 6)

Chapter 2

Background and Related Work

2.1 Existence of Code Cloning

There has been substantial amount of research conducted to understand the reasons behind the existence of clones in software. The literature attributes the following reasons that lead to the existence of similar code fragments in software.

- **Reuse**. Developers often fork the repository of existing solutions and further modify it to meet the requirements of new but similar functionalities [73]. This phenomena is not limited to forking the repositories but is well observed at other granularities e.g., copy-paste-modify of methods, classes, files, and packages, to name a few.
- **Maintenance Benefits**. Cloning offers many maintenance benefits such as separation of concern [125] and reusing an existing well-tested code [30]. It has been found that often clones diverge significantly from the original code fragment in terms of functionality [73]. In such situations, cloning offers *separation of concern*, a desirable design principle, where the original code and the duplicated code are to evolve at different paces. Moreover,

the cloned code brings in the benefit of having already been tested as the original code fragment, thereby, reducing the risk of introducing a new code that can break the system.

- **Language Limitations**.

  Certain programming languages lack sufficient abstraction mechanisms, e.g., *inheritance*, *generic types*, and *parameter passing*. This makes it difficult for developers to write a reusable code, which results into code duplication.

- **API Usage Protocols** Often developers need to adhere to a specific order in which the function calls are to be made while using certain libraries or APIs, resulting into accidental cloning.

- **Code Generation** Frameworks and development environments often generate *starter code*, enabling developers to expedite the implementation of functionalities. This causes the amount of boilerplate code becoming significantly large which results into unintentional clones [14].

- **Fast Paced Development** Certain software development practices require fast and frequent deliveries. Writing a reusable code under such circumstances is not only time-consuming but is also error prone. Developers often copy-paste-modify existing codes to meet time-pressed deadlines.

2.2 Applications of Clone Detection

Knowing where clones exist in, or across, systems leads to many useful applications:

- **Software Maintenance** The clones detected can be re-factored into reusable components, which often is considered as an improvement in the software design. Refactoring leads to lower maintenance costs as there are fewer instances in the code that need to be maintained. In a recent survey Chatterji et al. finds that clone management is useful for maintenance tasks and affects long term system quality [27].

- **Library Candidate Identification** Clone detection helps in identifying library candidates which in turn improves re-usability and lowers maintenance cost. For example, a code fragment which is cloned multiple times, proves its usability and therefore is a good candidate to be included in a library as an API. Kawrykow and Robillard use code similarity to find instances of code that is already available in the form of APIs, also known as API imitation [75]. In their study on ten Java projects, they found 400 cases of API imitations. They argue that developers could improve the quality of their software by removing such instances of API imitation.

- **Plagiarism Detection and Copyright Infringement** Some of the most popular and practical uses of clone detection are to identify plagiarized code and unauthorized usage of software (license violation) [2]. Software plagiarism happens when someone copies a code and uses it in a way that it hides the ownership of the copied code. In a famous case between Oracle and Google, Oracle accused Google of copying and using parts of code from twelve source files and 37 Java specification in their Java APIs in the Android operating system.

- **Reverse Engineering product line architecture** Identification of common components across multiple source code repositories can help in extracting the common features in a product and its derivatives [44, 56]. This reverse engineering process helps in creating more streamlined product pipeline.

- **Dataset Curation** Large datasets of source code often contain a large amount of clones at different granularities such as method, class, file, and project to name a few. Empirical studies conducted over such datasets could get adversely affected by the amount of duplicate information present in the datasets. A modern usecase of clone detection is to curate large datasets that are void of clones [99].

- **Facilitating Research in Software Engineering** Clone detection has been successfully used in different areas of research pertaining to software engineering. For example, software evolution analysis, aspect mining research, program comprehension, software provenance analysis and code search to name a few [17, 79, 99, 162].

2.3 Clone Detection Techniques and Tools

Clone detectors differ substantially in the underlying techniques and scope of application. In terms of technical approach used, one can find techniques that are token-based [94, 137, 153], metrics-based [81, 105], learning-based [131, 159], tree-based [15, 64], graph-based [43], and text-based [127]. A detailed description of these techniques can be found elsewhere [126]. The techniques that are most related to my work are described as follows.

- **Token-based Techniques** Token-based techniques use sophisticated transformations on sourcecode such as using lexical analyzers to generate a stream of tokens. These techniques compare tokens of code fragments in order to determine whether the two fragments are clones or not. Token-based techniques when used in a bag-of-words model, are demonstrated to be resilient in detecting near-miss Type III clones. Sajnani et al. demonstrated that token-based techniques can be scaled to large datasets containing millions of lines of code [137]. CCFinderX [72], SourcererCC [137], and CloneWorks [152] are examples of popular token-based clone detectors.

  SourcererCC [137] and CloneWorks [153] have shown good accuracy in detecting Type I to near-miss Type III clones. SourcererCC and CloneWorks create a global token frequency map of all the tokens in the target dataset. They then use this global token frequency map to create an inverted partial-index of code blocks. Clone detection is carried out by querying the partial-index. CloneWorks differs from SourcererCC by using a modified

Jaccard similarity metric in detecting clones, whereas the latter uses Overlap similarity metric.

- **Metrics-based Techniques** Metrics-based techniques compute and compare different software metrics of two code fragments in order to determine if the fragments are clones or not. Such techniques instead of comparing codes directly, compare these metrics. Two code fragments are considered clones when their corresponding metrics value are similar. Metrics-based techniques when applied to Type III clone detection suffer from high false positive rates and scalability issues.

  Keivanloo et al. [76] introduced a hybrid metrics-based approach to detect semantic clones from Java Bytecode. This approach utilizes four metrics: two numerical metrics which are Java type Jaccard similarity, and Method similarity; and two ordinal metrics which are Java type pattern and Java method pattern. Mayrand et al. [105] used Datrix tool to calculate 21 metrics for functions. They compare these metric values, and then, functions with similar metric values are identified as clones. The similarity threshold was manually decided. A similar technique is used by Patenaude et al. to find method level clones in Java projects [120]. These metrics are compared across four points of comparisons namely: Name, Layout, Expressions, and Control flow. Two functions are reported as similar for a point of comparison only if the absolute difference is less than or equal to the delta threshold defined for each metric in that point of comparison. The deltas were defined on the basis of metric definition and their knowledge of the distribution of that metric on large scale systems.

  Kontogiannis et al. [81] used five modified versions of well known metrics that capture data and control flow properties of program to detect clones in the granularity of begin – end blocks. They experimented with two techniques. The first one is based on the pairwise Euclidean distance comparison of begin-end blocks. The second technique uses clustering. Each metric is seen as an axis and clustering is performed per metric axis. Intersection of clusters results into the clone fragments.

- **Learning-based Techniques** In learning-based techniques, features are extracted from the code fragments. These features are then used to train a machine learning model to detect clones. These learning techniques can be either supervised or unsupervised. Like metrics-based techniques, these techniques suffer from scalability challenges. Usually, the features extracted represent the structural information of the code and lack in semantic information, which in turn leads to a high false positive rate.

  Davey et al. [31] attempt to detect clones by training neural networks on certain features of code blocks. They create feature vectors based on indentations of each line in the code-block, length of each line in the code-block, and frequency of keywords used in the code block.

  White et al. [159] present an unsupervised deep learning approach to detect clones at method and file levels. They explored the feasibility of their technique by automatically learning discriminating features of source code. They report the precision of their approach to be 93% on 398 files and 480 method-level pairs across eight Java systems.

  Sheneamer and Kalita use AST and PDG based techniques to generate semantic and syntactic features to train their model [141] to detect clones.

  In Wei and Li's approach [158], a Long-Short-Term-Memory network is applied to learn representations of code fragments and also to learn the parameters of a hash function such that the Hamming distance between the hash codes of clone pairs is very small.

  CCLearner [92] is another approach that leverages deep learning to classify clone and non-clone method pairs. In this work, authors have used the information about the usage of terms in source code (such as reserved words or identifiers) to build the feature vector for each method.

2.4 Evaluation of Clone Detection Techniques

Clone detection tools and techniques can be compared using several parameters. These parameters can also be seen as the desirable properties of clone detection approaches. In the following we list some of the well-known parameters used in the literature.

- **Recall.** Recall is the fraction of true positives that have been retrieved over the total amount of true positives in the dataset. An ideal clone detector should be capable of detecting all of the clone pairs in a given dataset.
- **Precision.** Precision is the fraction of true positives retrieved among the total retrieved instances.An ideal clone detector should contain no false positives in its output.
- **Scalability.** A good clone detector should be able to detect clones in a large dataset with reasonable memory usage.
- **Execution Time.** A good clone detector should complete the execution in a reasonable amount of time depending on the size of input.
- **Robustness.** A good clone detector should be able to detect clones of different types with high precision and recall.
- **Portability.** A good clone detector should have minimum dependency on the target platform or language and should be easy enough to adapt to various languages.

The effectiveness of clone detection tools is usually evaluated in terms of precision and recall. There has been good progress in measuring recall systematically of clone detection tools. Svajlenko et al. created BigCloneBench, a dataset of real world Java clones [146]. They also created BigCloneEval, a tool that estimates recall of clone detectors based on BigCloneBench dataset by measuring how many of the labeled clone pairs are included in the output of a clone detector [151].

Measuring precision of a clone detector, however, is not simple. To measure the precision, one can choose to manually validate all the clone pairs reported by a clone detector. This process is extremely time consuming and impractical as the number of clone pairs reported by a tool on a standard dataset like BigCloneBench is in millions. A more practical process is to estimate the precision by humanly validating a random and statistically significant sample of clone pairs. This is what researchers do to estimate the precision of clone detectors [131, 137, 148, 156]. In this process, after running a clone detector on a dataset and getting the clone pairs, a random and statistically significant sample set of these clone pairs is assigned to multiple judges for manual inspection. The judges examine each pair to decide if it is a true clone and/or what type of clone it is. When all sampled pairs have been validated by all judges, researchers aggregate the judges' decisions, usually by taking the majority vote, and report precision.

2.5 Chapter Summary

This chapter gave an overview of the research in the area of clone detection. The chapter looked into the reasons behind why clones exist in software systems. Cloning could be intentional or unintentional. Reusing existing code in the form of copy-paste-modify to meet time-pressed deadlines and to avoid introducing new bugs through new code are some of the reasons why developers make clones. On the other hand, clones may exist out of the limitations of language and protocols of API usage as well as through automatic code generators.

Clone detection has many applications which span both industry as well as research. Some of the popular applications of clone detection are: library candidate identification, plagiarism detection, copyright infringement detection, dataset curation and facilitating research in software engineering.

Over the years, many tools and techniques have been proposed to detect clones in software. In the literature, several techniques are proposed that are token-based, metrics-based, learning-based, tree-based, graph-based and text-based. These techniques are compared based on parameters such as: recall, precision, scalability, execution time, robustness and portability.

Chapter 3

Empirical Studies

The material in this chapter is based on our work in the following papers, and is included here with permission from both Springer and the ACM.

- V. Saini, H. Sajnani, and C. Lopes. Cloned and Non-Cloned Java Methods: A Comparative Study. Empirical Software Engineering, 23(4):2232–2278, Aug 2018. `https://doi.org/10.1007/s10664-017-9572-7`

  This work received the **Best Paper Award** at ICSME 2016.

- C. V. Lopes, P. Maj, P. Martins, V. Saini, D. Yang, J. Zitny, H. Sajnani, and J. Vitek. Déjàvu: A map of code duplicates on GitHub. in Proceedings of the ACM on Programming Languages, 1(OOPSLA):84:1–84:28, Oct. 2017. `http://doi.acm.org/10.1145/3133908`

  This work received the **Distinguished Artifact Award** at OOPSLA 2017.

This chapter presents empirical studies that we conducted to analyze software clones as well as the extent of cloning. Study 1 makes an extensive use of software metrics to analyze and compare cloned methods from the non-cloned methods written in Java software. This study gives a detailed description of the software metrics and categorizes them into three aspects of software quality, namely, complexity, modularity and documentation. The study helps to understand the characteristics of clone method, which are useful to detect code clones.

Study 2 explores the extent of cloning in a popular software repository hosting online service GitHub. The study was conducted for projects written in a set of popular programming languages, namely, Java, Javascript, C/C++, and Python. The study was led by Prof. Cristina Lopes and involves collaborations with Prof. Jan Vitek. The detection of clones in the Javascript projects and their analysis was led by Prof. Jan Vitek and his students. I contributed to this work by addressing some of the scalability challenges of SourcererCC, the clone detector used to detect clones in this study. I improved the execution time of SourcererCC, a desirable requirement to carry out clone detection at large scale. Further, I carried out clone detection in Java projects and contributed to their analysis. The learnings from this study helps in identifying the areas that need special attention to scale clone detection to large datasets.

3.1 Study 1. Do Cloned Methods Differ from Non-cloned Methods?

Traditionally, code cloning has been presented as a design error, or more broadly as a *code-smell* [42, 119]. It has been argued that code cloning adversely impacts software by forcing the tracking of all pieces of code when a certain fault is known to exist, or a new behavior is required, a task that can easily be error prone. Consequently, considerable effort has been put into clone detection [8, 16, 37, 38, 52, 66, 72, 84, 86], clone removal [80, 83] and clone refactoring [143].

Despite the disadvantages of code cloning, clones also leverage the difficulty that software development typically implies. They speed-up development and improve developer productivity [73], allow the reuse of tested code, or allow a better separation of concerns [73, 79].

These two different, but defensible, opinions raise the question of whether code cloning should be avoided, eliminated, tolerated or even recommended, and under what situations. The interest and the lack of knowledge on the topic led to a call to conduct more empirical studies about code clones [126]. Koschke has presented several important open issues [82], one of which being the relation of clones to quality attributes. To assess the impact of clones on defect occurrence of software products, Rahman et al. conducted a study on 4 projects. They did not find any evidence that cloning is harmful [121]. In another study conducted on 36 Java projects, Sajnani et al. found that cloned methods have lower defect density than that of the non-cloned methods [135].

Building on this series of studies, we conduct a statistical study to explore the relationship between code clones and 27 software quality metrics on 3,562 Java projects, totaling 1,029,309 methods. We note that this work is an extension of our work published earlier [133], where we analyzed 644,830 cloned methods and 842,052 non-cloned methods in 4,421 projects. We noticed an inconsistency in our earlier work which we have addressed here. Addressing the inconsistency did not alter the conclusions of the study; it simply reduced the number of methods and projects suitable for analysis. We explain the inconsistency in Section 4.3.1.

Also, in this work, we added a mixed-method analysis to gain further insights on the results obtained from the quantitative analysis. The mixed-method analysis, helped us identify the presence of methods that have no method-body (interfaces, stubs) in our dataset. To address this we removed the no method-body methods from the dataset and redid the quantitative analysis(Section 3.1.6.3). The conclusion of quantitative study, however, remains unchanged from the previous study.

3.1.1 Definitions and Thresholds

We consider a method to be a clone of another, if there exists a syntactically similar method (70% overlap similarity; please see Section 3.1.5.2 for more details) in the same project – we focus on intra-project clones only.

There could be various confounding variables like developer skills, project management styles, release cycles, and nature of projects(functionality, target users), etc,. that could affect how developers implement methods. For example, a team might enforce stricter regulations on the comments, size of methods, and design patterns. Another team might have more lenient regulations. If we mix the methods of the projects from these two teams, the overall distinction between the cloned and non-cloned methods might disappear. The distinction will be harder to make when we mix the methods from many different teams. However, if we analyze the methods from the projects separately, we can see how the cloned methods in a project differ form the non-cloned methods, if they differ. We chose to do intra-project analysis to address the above mentioned confounds. Most of the available clone-detectors are shown to reliably detect only Type-1, Type-2, and Type-3 clones; Therefore, in this study we exclude Type-4 clones.

In the statistical analysis, we conduct Wilcoxon Rank-Sum tests or use Linear Regression Models, and we consider *significant difference* to be a combination of the P-value and the effect size of the tests. Specifically, we are looking for P-values smaller than 0.05 (denoted $p < 0.05$) and the effect-size given by Cohen's d or Hedge's g greater than 0.1 (denoted $r > 0.1$).

FQMN: *FQMN* refers to Fully Qualified Method Name. Consider a method *getName* in a class *Student*, which is in a package named *com.foo.bar*. The FQMN of the method *getName* would be *com.foo.bar.Student.getName*.

3.1.2 Research Questions

We seek answers to the following research questions:

Research Question 1: *In a project, are cloned methods more or less complex than non-cloned methods?*

Unnecessary complexity has obvious disadvantages on a software solution and negatively affects most of its internal characteristics. If cloned code has higher code complexity than non-cloned code, the advantages that code cloning might introduce should be carefully balanced with the negative impact it has on maintainability, cost-efficiency, security and so on. We use 18 software quality metrics that address code complexity.

Research Question 2: *In a project, are cloned methods more or less modular than non-cloned methods?*

We define modularity as the ability of a certain piece of code to independently represent a specific functionality or semantic property. A method is said to be more modular if it has low dependency on variables, properties or methods which are defined out of its scope.

If cloned code is found to be less modular than non-cloned code, this suggests code cloning should be avoided. Modular artifacts are easier to maintain as their understanding does not depend on their contextual location in the source code.

We compute modularity using seven *Modularity* metrics.

Research Question 3: *In a project, are cloned methods more or less documented than non-cloned methods?*

The importance of well-documented source code is well known, but it is of further relevance in cloning code as programmers are expected to be more unfamiliar with code they simply copy-and-paste. A finding that relates code clones with minimal code comments, or simply with their nonexistence, will translate into a software artifact that is hard to understand and debug, comparing to a counterpart where the programmer developed the totality of the source code.

3.1.3 Findings

Overall we found no evidence that the quality metrics of cloned methods are significantly different than those of the non-cloned methods for the vast majority of the metrics. The metric that showed the most significant difference was size, measured here as the number of statements: on the dataset of 412,705 cloned and 616,604 non-cloned methods, we found that cloned methods are on an average (median) 18% smaller than the non-cloned methods.

A small fraction of the projects shows significant difference in the quality of cloned and non-cloned methods, even after controlled for size. Specifically, for three metrics out of the total of 27 approximately 30% projects contained cloned methods with significantly different values of the quality metric than non-cloned methods. For all the other 24 metrics, only a very small number of projects, or in some cases no projects at all, contained significant differences between clones and non-clones.

We performed a mixed method analysis to understand why the cloned methods differ from the non-cloned methods with respect to the size metric. We found one possible factor that may lead to the size differences– presence of many small auto-generated methods that are deemed clones because of their similar structure. Additionally, we gained some more insights about cloned methods. Specifically, among the clones that do not seem to be automatically generated, we found many instances of multi-purpose methods, i.e. methods that have very similar structure and vocabulary but do the opposite (expand vs. collapse, etc.). We also found that certain functionality tends to be more pervasive in cloned methods than non-cloned methods. Additionally, we manually evaluated the quality of comments and the comprehensibility of cloned and non-cloned methods but found no statistically significant difference.

Outline: The remainder of the paper is organized as follows. Section 3.1.4 discusses related work; Section 3.1.5 explains the study design and introduces the dataset, the clone detection technique and the software quality metrics used in the study; Section 3.1.6 presents and discusses the results; Section 3.1.7 presents the detailed mixed method analysis; the limitations of this study are discussed in Section 3.1.8; finally, we summarize our findings and discuss future work in Section 3.1.9.

3.1.4 Related Work

Code cloning and its impact has a long history as a topic of inquiry in the research community. In this section, we draw comparisons with existing research in this direction.

Traditionally, researchers have discussed many negative effects of code cloning [42, 67, 69, 87, 102, 161], however, the widespread presence of clones have motivated researchers to dig deeper and understand the usage scenarios.

Kapser et al. presented eleven patterns by examining clones in two projects [73]. They found out that not all usage patterns have negative consequences; in some cases, they identify patterns of cloning that they believe are beneficial to the quality of the system.

Ossher et al. looked at circumstances of file cloning in open source Java projects and classified the cloning scenarios into good, bad and ugly [117]. These scenarios included good use-cases like extension of Java classes and popular third-party libraries, both large and small. They also found ugly cases where a number of projects occur in multiple online repositories, or have been forked, or were copied and divided into multiple sub projects. From a software engineering standpoint, some of these situations are more legitimate than others.

Kim et al. studied clone evolution in two open source projects. They found that most of the clone pairs are short lived and about 72% of the clone pairs diverge within eight commits in the code repository [79]. They found that several clones exist by design and cannot be refactored because of the limitation of programming language or it would require a design change. Similarly, de Wit et al. proposed CLONEBOARD, an Eclipse plug-in implementation to track live changes to clones and offering several resolution strategies for inconsistently modified clones [33]. They conducted a user study and found that developers see the added value of the tool but have strict requirements with respect to its usability.

Cordy analyzed clones and intentions behind cloning of a financial institution system and argued that external business factors may facilitate cloning [30]. He mentioned that financial institutions avoid situations that can break the existing code under any circumstances. Abstractions might introduce dependencies and modifying such abstractions induces the risking of breaking existing code.

Cloning minimizes this risk as code is maintained and modified separately localizing the risk of errors to a single module. Similarly, Rajapakse et al. found that reducing duplication in a web application only had negative effects on the modifiability of an application [122]. They noted that after significantly reducing the size of the source code, a single change required testing of a vastly larger portion of the project.

Rahman et al. investigated the effect of cloning on defect proneness on four open source projects [121]. They looked at the buggy changes and explored their relationship with cloning. They did not find evidence that cloned code is riskier than non-cloned code. This finding is also supported by Sajnani et al. [135], where they conducted an empirical study comparing the presence of bug patterns in cloned and non-cloned code on 36 open source Java projects. Surprisingly, they found that the defect density of cloned code is less than 2 times the defect density of the non-cloned code.

Islam and Zibran conducted a similar study where they compared the vulnerabilities, detected using static source code analyzer named *PMD*, of cloned and non-cloned code in 97 software systems. They found no statistically significant difference between the vulnerabilities of cloned and non-cloned code [61].

Brutnik et al. used clone detection techniques in a novel way to find cross-cutting concerns in the code [24]. They manually identify five specific cross-cutting concerns in an industrial C project and analyze to what extent clone detection is capable of finding them. The initial results were positive.

Many researchers have assessed the impact of cloning on software maintenance by studying the stability of cloned and non-cloned code. One key point driving these studies is to know if cloned code is more stable than the non-cloned code during software evolution.

Mondal et al. compared the stability of cloned and non-cloned code in 12 software systems written in three languages, Java, C, and C#. They reported clones of Type-1 and Type-2 make the system more unstable when compared to the non-cloned code. Clones of Type-3, however, are more stable than the non-cloned code [108].

Krinke also compared the stability of cloned code to the stability of non-cloned code. Krinke analyzed 200 weeks of evolution of five software system and found that cloned code, in general, is more stable than the non-cloned code [88]. Krinke reported that when additions are made to a system, the additions are made more often in the non-cloned code. However, when only deletions are considered, cloned code was found to be less stable than non-cloned code.

Gode and Harder replicated and validated Krinke's study, where they used an incremental clone detection technique. They found similar results and reported cloned code to be more stable than non-cloned code in general [46]. In another study [48], Gode and Koschke studied clone evaluation in three software systems written in Ada, C, and Java. They reported that most clones are rarely changed and the number of unintentional inconsistent changes to clones is small.

Hotta et al. investigated how much the presence of duplicate code is related to software evolution. To this end, they compared the maintainability of duplicate code and non-duplicate code on 15 open source software systems. Their result showed that the presence of duplicate code does not have a negative impact on software evolution [59].

In another study, Lozano and Wermelinger compared maintenance effort on cloned and non-cloned code [101]. They used statistical and graphical analysis to report that having a clone may increase the maintenance effort of changing a method. They further report that the maintenance effort seems to increase depending on the percentage of the system affected whenever the methods that share the clone are modified.

Thus, evaluating the positive and negative impacts of cloning has been a continuous balancing act. Like some of the recent studies, our study also presents one more piece of evidence that cloning may not be that bad after all. However, our study differs from similar studies in the literature in several aspects.

We explore the relationship between a set of 27 well known software metrics across three categories and code clones, and compare and contrast the nature of this relationship in non-cloned code. These metrics are a proactive way of looking at the patterns which are potential threats for the code base. Other studies have looked at the relationship of bugs with cloning by mining the version control systems to find out the bugs associated with clones. While such reported bugs provide a measure of external quality of the overall project, researchers have identified several issues with respect to completeness and correctness of such data. Moreover, like all complex problems, the issue of code cloning being bad or not will only be fully understood by looking at it from several angles and with several methodologies. Moreover, several studies have also evaluated the effectiveness of software quality metric systems and found it to be a useful indicator of code quality [11, 70, 93].

We conduct this study on 3,562 open source Java projects and compare cloned methods to non-cloned methods with respect to 27 software metrics. This scale not only helps us to instill more confidence in the findings of the study, but also helps to discover and hence account for confounding factors that might impact the validity of the study. The findings of this study not only help in understanding the nature of code clones, but also help in identifying hot spots which are more likely to be re-used in the future.

3.1.5 Study Design

As shown in Figure 3.1, the overall design of our study is a two step process, with both steps being merged in a model from which we derive our results. In the first step, we compute the software quality metrics for every method in each project. This step gives us a map to associate methods with their software quality metric values. In the second step, we use a clone detection tool to identify all the intra-project clones present in each project, which results in a map that contains the information of whether a method has a clone or not.

Combining the result of the previous two steps, we have, for each method in a project:

1. Whether the method has a clone or not; and
2. Software quality metric values for the method

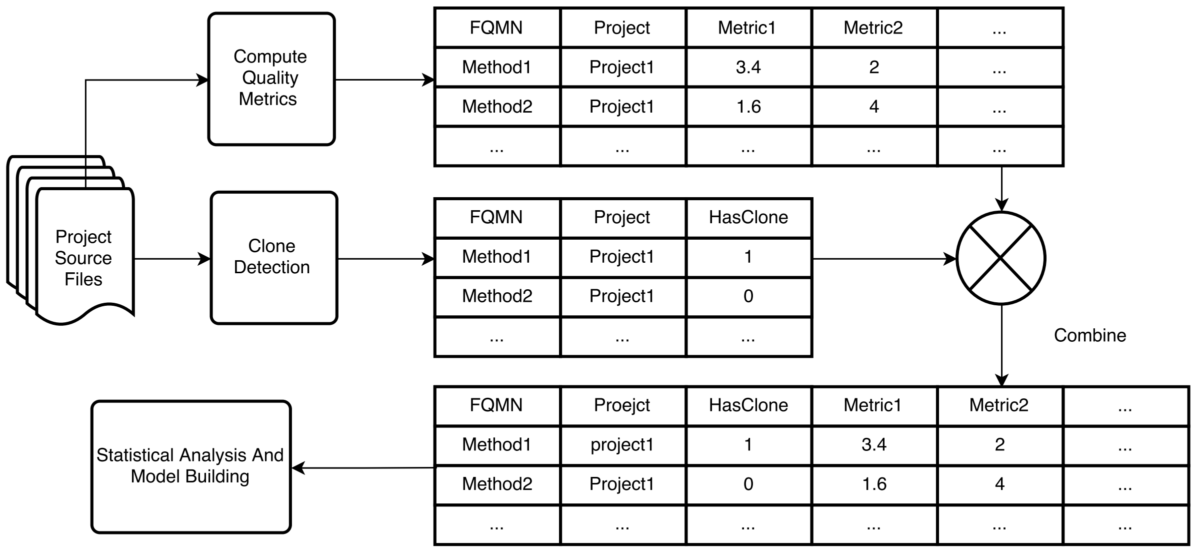
After these steps we had all the information needed to start a statistical analysis and answer the research questions. In the remainder of this section we describe each aspect of our study design in detail.

3.1.5.1 Dataset

The dataset used in this study consists of 3,562 Java projects hosted by Maven [104]. The comprehensive list of projects with their version information can be found at `http://mondego.ics.uci.edu/projects/clone-metrics/`. These projects are of varied size and span across various domains including but not limited to search and database projects, server projects, distributed systems, machine learning and natural language processing libraries, and network systems. The large variety of projects was chosen to avoid any bias on our results towards a certain project characteristic, as they represent a wide range of sizes and domains.

Dataset Curation:

We downloaded the entire Maven repository available at UCI Source Code Data Sets [97]. We then selected the top versions of all the projects that have their source files with them. This gave us 18,852 projects. We observed that there were many empty projects in this dataset; so in order to remove very small or empty projects we filtered out all the projects which have less than 100 Java statements. This resulted into a total of 12,872 projects. On every project we executed SourcererCC, a token based clone detector tool, and JHwak, a Java metric tool. We executed these tools at method level granularity. For each project, we joined (inner join) the output of these tools based on the fully qualified method names (FQMN), which is a common filed in the output of the two tools. The join works well for the methods which have unique FQMN; but the overloaded methods in a Java class have the same FQMN, and therefore, the inner join produces multiple entries for the overloaded methods. This produces inconsistent results for overloaded methods which could negatively affect the result of our study; so we removed all the overloaded methods from our dataset. We note that this inconsistency is present in our previous paper [133]. As a final filtration step, we rejected the projects that have less than or equal to 10 clone or 10 non-cloned



FQMN	Project	Metric1	Metric2	...
Method1	Project1	3.4	2	...
Method2	Project1	1.6	4	...
...	...	...	...	...

FQMN	Project	HasClone
Method1	Project1	1
Method2	Project1	0
...	...	...

FQMN	Proejct	HasClone	Metric1	Metric2	...
Method1	project1	1	3.4	2	...
Method2	Project1	0	1.6	4	...
...	...	...	...	...	...

Figure 3.1: Study Design
(FQMN is short for Fully Qualified Method Name).

methods, resulting into 3,562 projects. We applied this filter for two reasons:

- To do a meaningful empirical study of cloned and non-cloned methods we wanted to have only those projects which exhibit some extent of cloning; and
- The filter made sure that we consider only the projects that have at least 20 methods (at least 10 non-cloned and 10 cloned methods) in them. Smaller projects may not have many methods/features to clone from. Moreover, a large number of such projects could negatively affect the study.

The top most plot on Figure 3.2 describes the size distribution of the projects in our dataset. The X-axis represents the binned number of Java statements (NOS). The Y-axis represents the percentage of projects.

3.1.5.2 Code Cloning

We use our own scalable token-based clone detection tool, SourcererCC [137]. Sourcerer-CC is capable of detecting Type1, Type2, and Type-3 method level clones. It was developed with

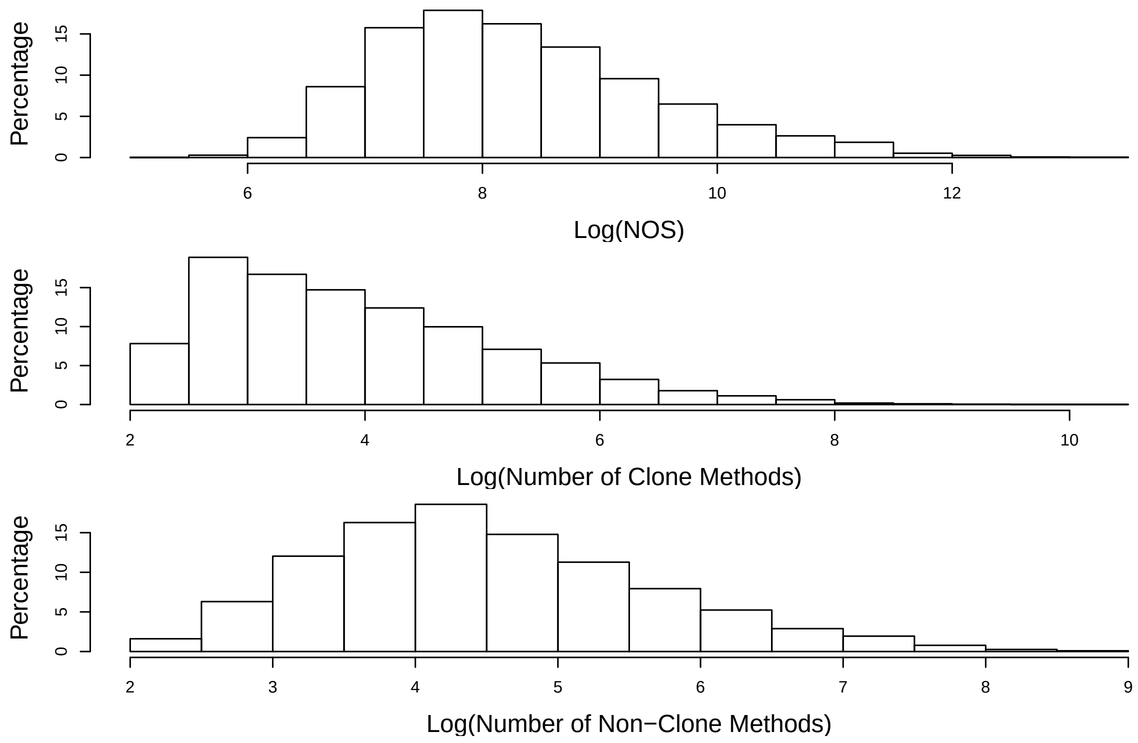


Figure 3.2: Size distribution of projects. Top, the number of Java Statements (NOS). Middle, the binned number of cloned methods. Bottom, the binned number of non-cloned methods in log scale. The Y-axis shows the percentage of projects. All charts are in log scale.

Table 3.1: BigCloneBench Recall Results

Tool	T1	T2	VST3	ST3	MT3	WT3/T4
SorcererCC	**100**	98	93	61	5	0

scalability as a main concern, and uses a heuristic-optimized index of tokens to significantly reduce the number of code-block comparisons needed to detect the clones, as well as the number of required token comparisons needed to judge a potential clone. Testing has proved the tool to scale to a large repository with 250M lines of code in a single workstation (3Ghz i7 CPU, 12Gb RAM), and has the state of the art precision and recall[1]. The authors tested the recall of SourcererCC on two different datasets: i) Mutation Framework [128] and ii) BigCloneBench [147].

SourcererCC shows 100% recall on Mutation Framework, which shows it can handle many, if not all types of edits developers make on cloned code.

The recall results for SourcererCC, measured on BigCloneBench are summarized in Table 3.1. SourcererCC shows 100% recall for Type-1 clones, 98% recall for Type-2 clones, 93% recall for *VST3* clones, 61% recall for *ST3* clones, 5% recall for *MT3* clones, and 0% recall for *WT3/T4* clones. The performance of SourcererCC matches the performance of other competing tools. We encourage readers to see the performance of other competing tools elsewhere [137].

The precision of SourcererCC was measured on the clone pairs that it reported on the BigCloneBench dataset. To this end, authors reported precision at two different configurations: i) the minimum size of the methods is set to 6 NLOC, and ii) the minimum size of the methods is set to 10 NLOC. They reported the precision to be 83% at 6 NLOC configuration and 86% at 10NLOC configuration. Please refer the original paper for more details.

We chose the following configuration to run the SourcererCC: threshold value for similarity was set to 70%, meaning methods with similarity '70% or more' were reported as clones by

[1]Specific hardware details and testing conditions are in the original article.

the tool.

SourcererCC uses overlap similarity to compute similarity between two methods. The overlap similarity measure simply computes the intersection between the code fragments by counting the number of tokens shared between them.

In addition to removing the overloaded methods, for the reason explained in Section 3.1.5.1, we also removed the methods with less than 25 tokens to ignore empty methods, stubs, method contracts defined in interfaces, abstract classes and other interferences. At this configuration, SourcererCC has demonstrated to have a precision of 83% [137].

In total we found 412,705 cloned and 616,604 non-cloned methods in our dataset of 3,562 projects. Figure 3.2 shows the distribution of cloned methods in the projects (middle plot). The X-axis shows the binned number of cloned methods in log scale. The Y-axis shows the percentage of projects. The bottom plot in the same figure shows the similar information but for non-cloned methods.

The median size, measured in lines of code (NLOC), for cloned and non-cloned methods is 13 and 16 respectively. The median size, measured in number of statements (NOS), for clones and non-cloned methods is 9 and 11 respectively (in Section 3.1.5.3 we provide details about these metrics).

Figure 3.3 shows the size distribution of all methods (top 2); cloned methods (middle 2); and non-cloned methods (bottom 2). The X-axis of the three histograms, show the binned size of methods measured in NLOC in log scale. The Y-Axis shows the number of methods in each bin.

Precision: In order to measure the precision of SourcererCC on our dataset, We randomly selected 401 clone pairs for manual inspection. This is a statistically significant sample with a 95% confidence level and a ±5% confidence interval. The sample set was inspected by 3

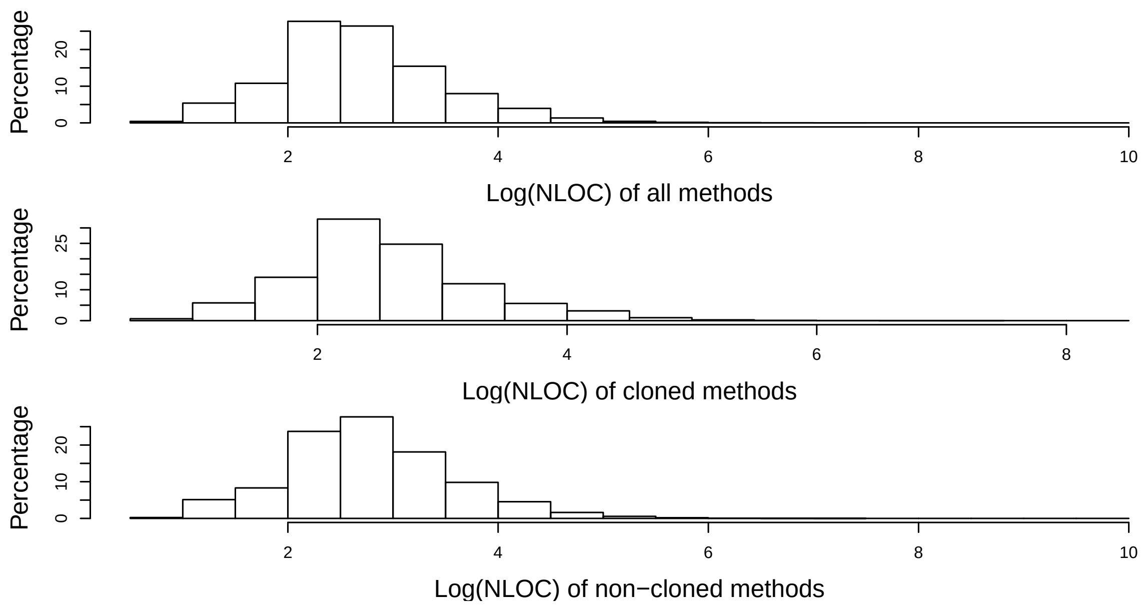


Figure 3.3: Distribution of methods measured using number of lines of code metric.

judges to figure out the true and false positives. All of the judges have 7+ years of open source software development experience; one has 5+ years of industrial software development experience; and two of them have 3+ years of industrial software development experience. In the cases where there were conflicts, we resolved them by voting. We note that one of the judges is also an author of this paper.

Judge 1 found all 401 as true positives; Judge 2 found 394 as true positives and 7 as false positives; and Judge 3 found 395 as true positives and 6 as false positives. After voting, only one clone-pair was declared as a false positive and the rest 400 were deemed true positives, resulting into the high value of 99.7% precision for SourcererCC. The precision falls to 97%, if a clone pair is considered as a false positive when any of the judges vote it as a false positive.

3.1.5.3 Software Quality Metrics

All the metrics used were calculated at the method-level using version 6 of the JHawk tool [63]. JHawk has been widely used in academic studies on Java metrics [3, 5, 6, 7, 20,

Category		Name	Description
Code Complexity	↓	VREF	Number of variables referenced
	↓	VDEC	Number of variables declared
	↓	TDN	Total depth of nesting
	↓	NOS	Number of statements
	↓	NOPR	Total number of operators
	↓	NLOC	Number of lines of code
	↓	NEXP	Number of expressions
	↓	NAND	Total number of operands
	↓	MDN	Method, Maximum depth of nesting
	↓	LOOP	Number of loops (for,while)
	↓	HVOL	Halstead volume
	↓	HVOC	Halstead vocabulary of method
	↓	HLTH	Halstead length of method
	↓	HEFF	Halstead effor to implement a method
	↓	HDIF	Halstead difficulty to implement a method
	↓	HBUG	Halstead prediction of number of bugs
	↓	COMP	McCabes cyclomatic complexity
	↓	CAST	Number of class casts
Modularity	↓	XMET	External methods called by the method
	↓	NOA	Number of arguments
		MOD	Number of modifiers
	↓	LMET	Local methods called by the method
	↓	EXCT	Number of exceptions thrown by the method
	↓	EXCR	Number of exceptions referenced by the method
	↓	CREF	Number of classes referenced
Documentation	↑	NOCL	Number of comment lines
	↑	NOC	Number of comments

Table 3.2: Software Quality Metrics

22, 51, 57, 103, 110, 111, 114, 139].

To evaluate quality of the methods, we use the totality of the 27 software quality metrics supported by JHawk, and that can be seen in Table 3.2. We divide the metrics into 3 groups: (i) metrics assessing code complexity; (ii) metrics assessing modularity properties; and (iii) metrics assessing how well the code is documented. Next we describe each of these categories:

Code Complexity Metrics. These metrics capture the complexity of the code itself. We used metrics as they are the standard technique for analyzing software. For example, a set of metrics used is derived from the metrics included in SQO-OSS, specifically designed for evaluating the quality of open source projects [138]. SQO-OSS is composed of well-established and validated software quality metrics, which can be computed either from source code or from surrounding community data.

Moreover, Nagappan et al. conducted a study where many software metrics including but not limiting to *number of executable lines* in a function, *number of parameters* in a function, *number of methods called* by a function or *number of global variables written/read* in a function etc., were shown to correlate with the post-release defects of software [113]. Several studies have also evaluated the effectiveness of software metric projects and found it to be a useful indicator of code quality [11, 70, 93].

- *VREF* counts the total number of variables referenced in a method. If a method has a large number of arguments, it declares a large number of variables, or it references a large number of variables then, it may be a sign that the method is trying to do too much. It also means that the changeability (tendency to change) of the method is high.
- *VDEC* is the total number of variables declared in a method.
- *NOS* measures the number of Java statements in a method. A statement forms a complete

unit of execution and usually is terminated with a semicolon (;). A statement can be spread over multiple lines; for example one can decide to start the curly brace of an *if statement* either from the current line or from the next line.

- *NOPR* measures the total number of operators (arithmetical operators, method names) used in a method.

- *NLOC* measures the total number of lines of code in a method.

- *NEXP* measures the total number of expressions in a method. Oracle notes that "*An expression is a construct made up of variables, operators, and method invocations, which are constructed according to the syntax of the language, that evaluates to a single value*". A higher value of number of expressions per statement may indicate a complex method.

- *NAND* measures the total number of operands (variables, numeric and string constants) used in a method.

- *MDN* is the measure of maximum depth of nesting in a method. A higher depth of nesting in a method indicates higher complexity of the method.

- *HVOL*, Halstead volume, is computed using *HLTH* and *HVOC* and can be seen as a measure of how much information does a reader need to absorb to understand a method. A small number of statements with a high Halstead Volume would suggest that the individual statements are quite complex.

- *HVOC*, Halstead vocabulary is computed as the sum of unique operators and unique operands in a method. It gives a sense of the complexity among the statements – for example a method which repeatedly uses a small number of variables is less complex than the method which uses a large number of different variables.

- *HLTH*, Halstead length [54], is the most basic of the Halstead metrics, which simply is the sum of *NOPR* and *NAND*.

- *HEFF*, Halstead effort, attempts to estimate the amount of work that a developer would take to recode a particular method.
- *HDIF*, Halstead Difficulty, uses a formula based on the number of unique operators, the number of unique operands, and the total number of operands to assess the complexity of a method. It suggests how difficult the code is to write and maintain.
- *HBUG*, Halstead bugs, attempts to estimate the number of bugs in a method.
- *COMP* measures the Cyclomatic complexity of a method. It is the standard metric introduced by McCabe in 1976 [106].

For all these *Complexity* metrics, a higher value would mean higher code complexity, therefore low values are desirable.

Modularity Metrics. It is another category of metrics, that pertains to encapsulation and modularity of software.

- *XMET* measures the total number of external methods called by a method. External methods are the methods that belong to a class other than the class in which the caller method is defined.
- *NOA* measures the total number of arguments expected by a method's signature. A higher number of arguments could mean that a method has higher dependency and lower modularity.
- *MOD* is the total number of modifiers (public, private, abstract, static etc.) applied to the signature of a method.
- *LMET* totals the number of local methods called by a method.
- *EXCR* and *EXCT* measure the total number of exceptions referenced and the total number of exceptions thrown by a method respectively.

- *CREF*, number of classes referenced, measures the total number of external classes referenced by the method.

For all of these *Modularity* metrics except for *MOD*, a higher value would mean that the code is less modular and therefore low values are desirable. We cannot find in literature what should be the desirable value for the *MOD* metric.

Documentation Metrics. These metrics capture properties of code-as-text.

- *NOCL* counts the total number of comment lines in a method.
- *NOC* measures the total number of comments in a method. It is different from *NOCL* as *NOC* counts a comment spread over, say 3 lines, as 1. Whereas *NOCL* will count it as 3.

A higher value for these metrics would mean that the code is documented more. A more documented code is easier to follow and therefore a higher value for these metrics is more desirable, purely from the quantitative point of view. However, redundant and useless comments can reduce the readability of code too. Therefore a qualitative analysis (see Section 3.1.7.1) is needed to gain more insights.

Most of the *complexity* and the *modularity* metrics (*COMP*, *HLTH*, *HVOC*, *HVOL*, *HDIF*, *HEFF*, *NOPR*, *NAND*, and *HBUG*) are the standard software quality metrics suggested by McCabe [106], and Halstead [54].

The metrics *CREF*, *XMET*, and *LMET* are derived from the object-oriented (OO) metrics that were suggested by Chidamber and Kemerer [29]. Basili et al. found that OO metrics appeared to be useful for predicting defect density [12]. Subramanyam and Krishnan conducted a survey on eight more empirical studies and showed that OO metrics are significantly associated with defects [145].

Additionally, some of these metrics (*NOS*, *MDN*, *TDN*, *CREF*, *XMET*, *LMET*, *HLTH*, *NOC*, *NOCL*, *CREF*, *XMET*, *LMET*, *NEXP*, *NLOC*) are also part of the SQO-OSS quality model.

Table 3.2 shows an additional piece of information for each metric. First, an up or down arrow near the metric name indicates what "better" means for that specific metric; this judgment comes either from studies that have been reported in the literature or from the intuitions that drove the definition of the metric in the first place. When those judgments cannot be made, there is no arrow.

3.1.6 Study Results

In this section, we present the results of our analyses of code clones and their relationship with each of the three software quality metric categories.

RQ1: In a project, are cloned methods more or less complex than non-cloned methods?

We carried out two one-sided Wilcoxon Rank-Sum tests for each subject project and *Complexity* metrics. One Wilcoxon Rank-Sum test tested the hypothesis that cloned methods have lower (better) complexity than the non-cloned methods. The other test tested the hypothesis that non-cloned methods have lower (better) complexity than the cloned methods. The summary of the two tests are shown in the Figure 3.4. The vertical axis shows the metrics. A number inside the parenthesis shows the number of projects for which we observed statistically significant test results ($p<0.05$) and also at least a small effect size (Cohen's d; $r > 0.1$). The horizontal axis shows the percentage of these projects where we observed lower metric values for clones or non-clones.

To understand the chart consider an example of *CAST* metric. *CAST(1286)* means that for the metric *Cast* there were 1,286 projects with significant difference ($p<0.05$ and $r>0.1$) in the cloned and non-cloned methods. The horizontal light grey bar shows that in around 60%

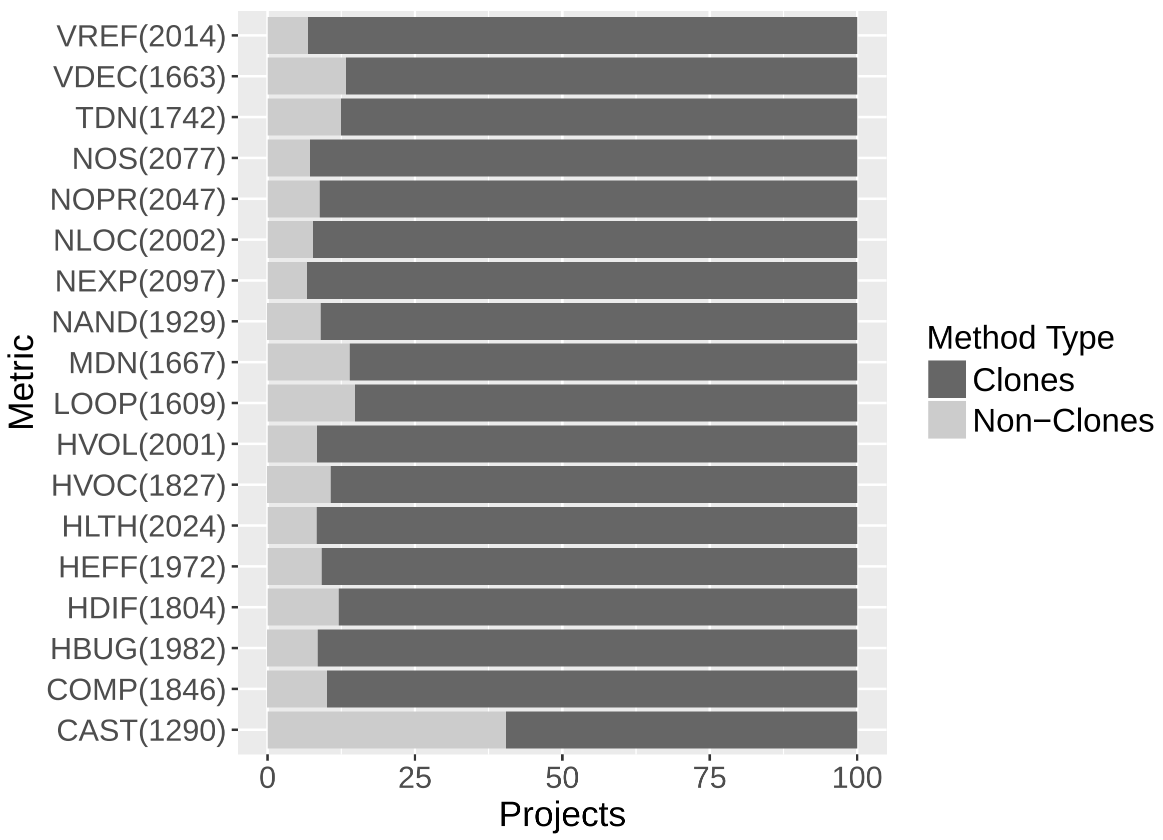


Figure 3.4: Wilcoxon Rank-Sum test: stacked bar chart comparing cloned and non-cloned methods.The Vertical-axis shows the *Complexity* metrics along with the number (in parenthesis) of subject projects that showed significant difference between cloned and non-cloned methods ($p<0.05$, $r>0.1$).

of these 1,286 projects cloned methods have lower $CAST$ metric values than the non-cloned methods. The dark grey bar, on the other hand shows that in around 40% of these 1,286 projects non-cloned methods have lower $CAST$ values.

To summarize, out of 3,562 projects, 1,290 to 2,097 projects showed significant difference ($p<0.05$ and $r>0.1$) in the complexity of cloned methods to that of the non-cloned methods. For all of the metrics more than 60% of the projects showed that cloned methods have lower complexity metric values than the non-cloned methods.

RQ2: In a project, are cloned methods more or less modular than non-cloned methods?

We carried out two one-sided Wilcoxon Rank-Sum tests for each subject project and *Modu-*

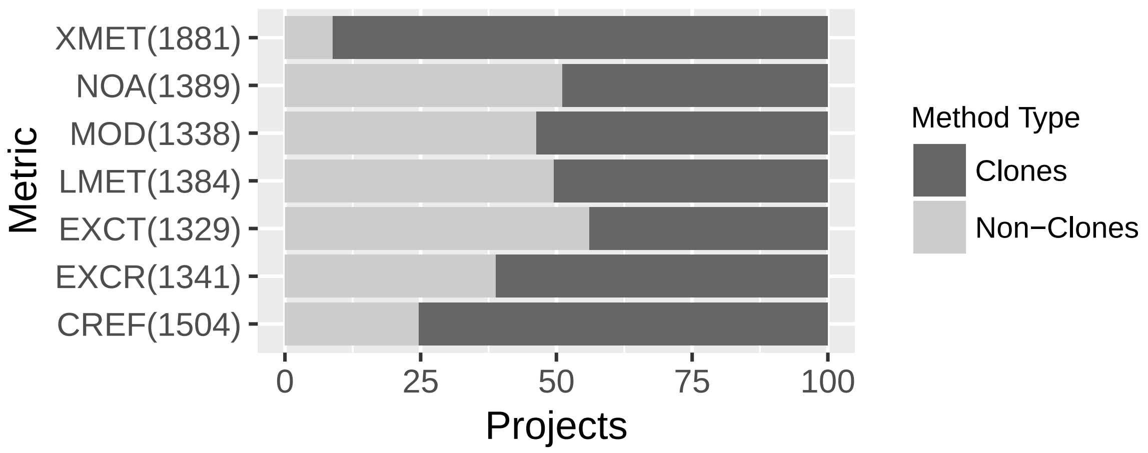


Figure 3.5: Wilcoxon Rank-Sum test: stacked bar chart comparing cloned and non-cloned methods.The Vertical-axis shows the *Modularity* metrics along with the number (in parenthesis) of subject projects that showed significant difference between cloned and non-cloned methods ($p<0.05$, $r>0.1$).

larity metrics. One Wilcoxon Rank-Sum test tested the hypothesis that cloned methods have lower (better) modularity metric values than the non-cloned method. The other test tested the hypothesis that non-cloned methods have lower (better) modularity than the cloned methods. The summary of the two tests are shown in the Figure 3.5; the axis description is similar to that of the Figure 3.4.

Out of 3,562 projects, 1,329 to 1,881 projects showed significant difference ($p<0.05$, $r > 0.1$) in the modularity metrics of cloned methods to that of the non-cloned methods. In more than 50 % of the projects, the cloned methods performed better on *XMET*, *MOD*, *EXCR*, and *CREF* metrics. Whereas, the non-cloned methods performed better on the *EXCT* metrics. For *LMET* and *NOA* metrics, we have almost equal number of projects voting for both cloned and non-cloned methods.

RQ3: In a project, are cloned methods more or less documented than non-cloned methods?

Like the above two RQs, we conducted two one-sided Wilcoxon Rank-Sum tests for the two *Documentation* metrics. For *NOC*, 1,777 projects showed significant difference ($p<0.05$, r

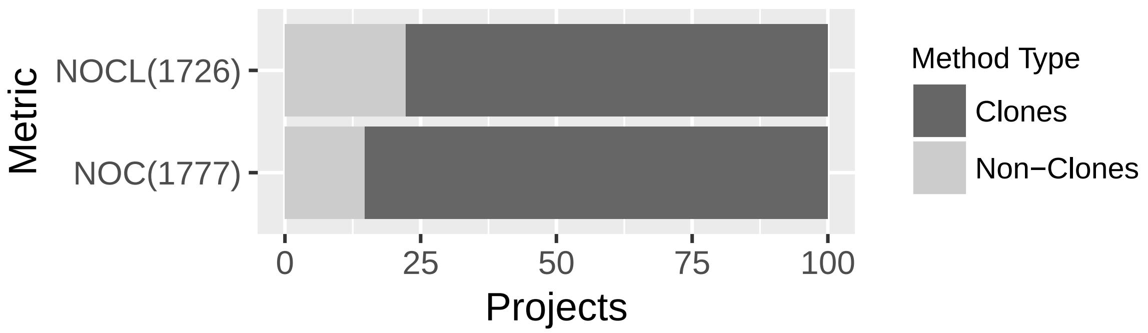


Figure 3.6: Wilcoxon Rank-Sum test: stacked bar chart comparing cloned and non-cloned methods.The Vertical-axis shows the *Documentation* metrics along with the number (in parenthesis) of subject projects that showed significant difference between cloned and no-cloned methods ($p<0.05$, $r>0.1$)

>0.1). For *NOCL*, 1,726 projects showed significant difference in cloned and non-cloned methods.

The summary of the results for these tests are shown in Figure 3.6. For both of these metrics, more than 70 % of the projects showed that the cloned methods have fewer code comments than the non-cloned methods.

From the above tests, for the projects where a significant difference was observed between clones and non-clones, most of the projects showed that the cloned methods are more modular and less complex than the non-cloned methods. Of the three quality metric categories, *Documentation* is the only one which showed negative results for cloned methods in comparison to the non-cloned methods.

Controlling for size:

Many observable program features correlate strongly with code size. This knowledge has been used pervasively in quantitative studies of software through practices such as normalization on size metrics [23, 26, 39, 136].

On an average (median) there are 34 cloned methods and 75 non-cloned methods in a project. The absolute median difference in the number of cloned and non-cloned methods in the projects is 41, which could negatively impact the study. To address this, we randomly selected an equal number of cloned and non-cloned methods from each project. For example, consider a project S, with C1 number of cloned methods and C2 number of non-cloned methods, such that C1 < C2. In order to match the number of cloned methods with non-cloned methods, we randomly selected C1 out of C2 non-cloned methods. Similarly, consider a project S2, with C1 number of cloned methods and C2 number of non-cloned methods, such that C2<C1. In this case, we randomly selected C2 number of cloned methods from the original pool of C1 cloned methods.

The median NLOC of cloned methods and non-cloned methods are 13 and 16 respectively. Also, the median NOS of cloned methods and non-cloned methods are 9 and 11 respectively, suggesting that on an average, the cloned methods are 18.18% smaller than the non-cloned methods. In order to make sure that the size is not impacting our study, we decided to redo our experiments using two size control strategies: i) dividing each metric by size; and ii) regression. We explain each of these strategies and the results below.

3.1.6.1 Dividing each metric by the size metric- NOS

In this strategy, for every method we divide each of the metrics with the method's *NOS* metric. For example, let us say for a method m, we have its *COMP* metric value to be 10 and its *NOS* metric value to be 50. Then the new *COMP* value of m becomes 10/50=0.2.

Also, for each project, we matched the number of cloned methods to the non-cloned methods. Two one-sided Wilcoxon Rank-Sum Tests were then carried out for each of the 3 categories of metrics. The summary of the results of these tests are shown in Figures 3.7 to 3.9.

These figures paint a different picture from the non-size-control experiments. Unlike the non-size-control experiments, the cloned-methods do not appear to win, in-fact for most of the metrics in the *Modularity* group of metrics, the bars for non-clones are longer than the bars

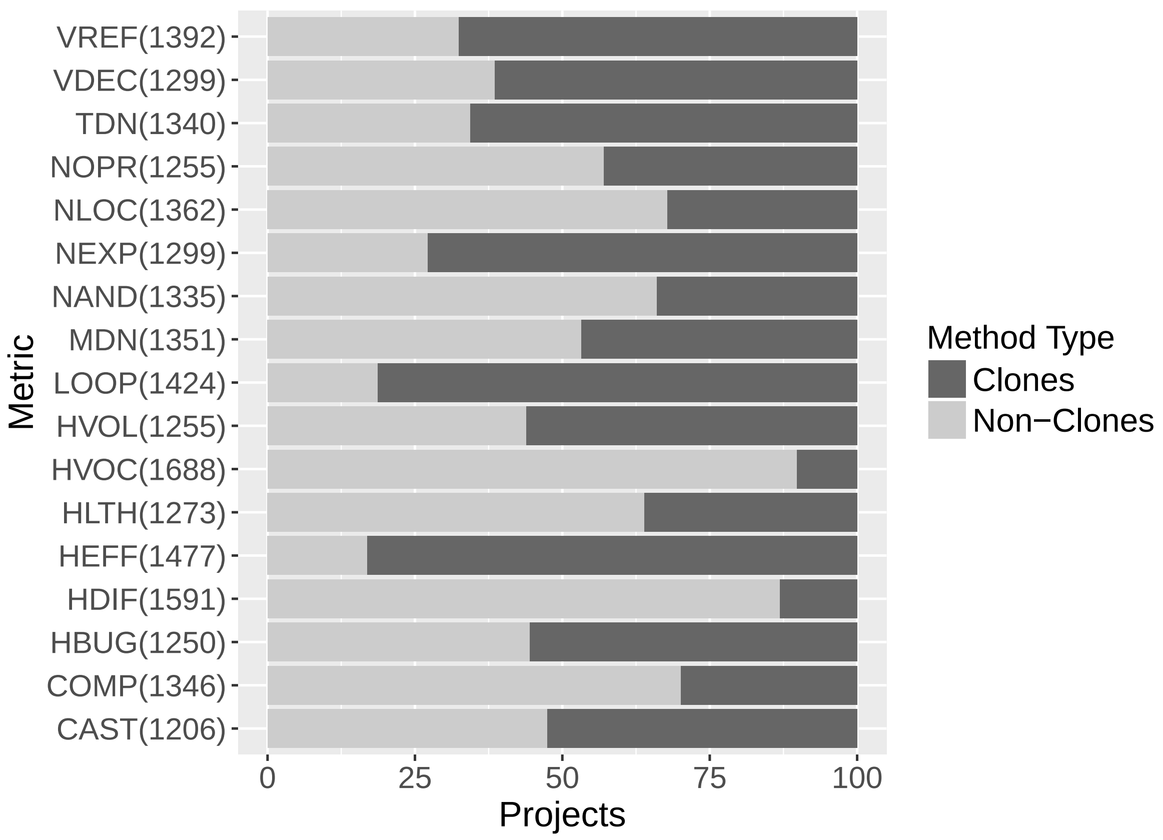


Figure 3.7: Wilcoxon Rank-Sum test: stacked bar chart comparing cloned and non-cloned methods.The Vertical-axis shows the normalized *Complexity* metrics along with the number (in parenthesis) of subject projects that showed significant difference between cloned and non-cloned methods (p<0.05, r>0.1).

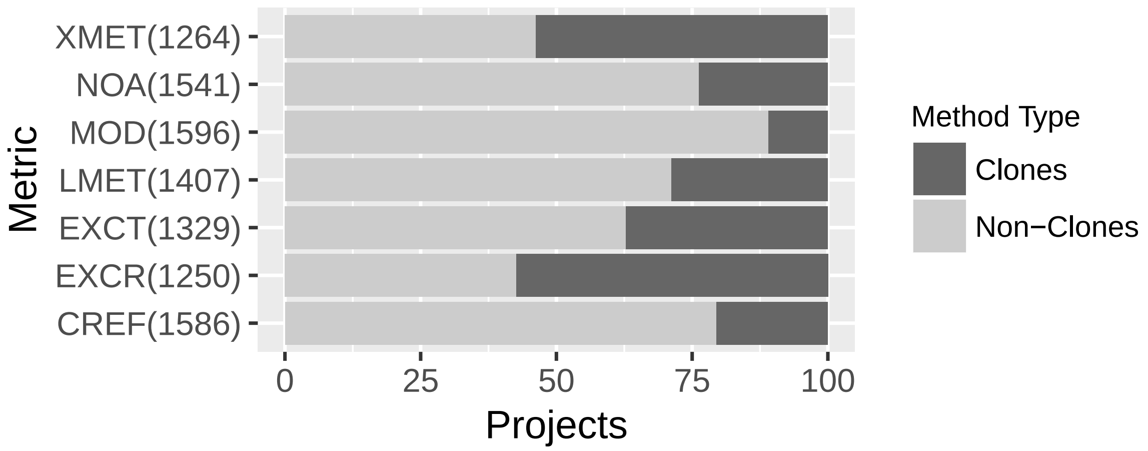


Figure 3.8: Wilcoxon Rank-Sum test: stacked bar chart comparing cloned and non-cloned methods.The Vertical-axis shows the normalized *Modularity* metrics along with the number (in parenthesis) of subject projects that showed significant difference between cloned and non-cloned methods (p<0.05, r>0.1).

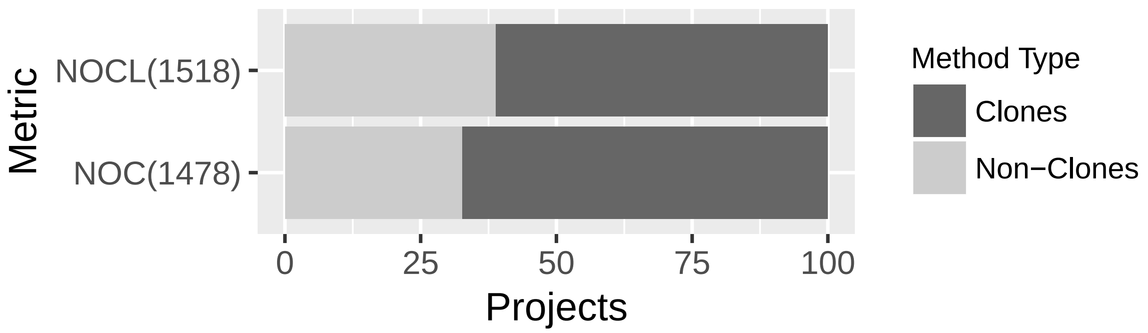


Figure 3.9: Wilcoxon Rank-Sum test: stacked bar chart comparing cloned and non-cloned methods.The Vertical-axis shows the normalized *Documentation* metrics along with the number (in parenthesis) of subject projects that showed significant difference between cloned and non-cloned methods ($p<0.05$, $r>0.1$).

for clones, signifying that the non-clones have better metric values than the cloned methods. In the *Complexity* group of metrics, both clones and non-clones have performed better on 50% of the metrics. In the *Documentation* metrics group, the clones have performed worse on both *NOC* and *NOCL* metrics.

A recent study on Java projects by Lopes and Ossher shows that dividing a metric by size is not a best way to control for size [100]. They found that certain characteristics of programs vary disproportionately with program size, sometimes even non-monotonically. This could mean that dividing each metric by the size metric might not be making the metrics independent of size. To verify this, we plotted the scatter plots between the quality metrics and the size metric- *NOS*, both before and after normalization of the quality metric. We observed for many cases that the normalization of metrics does not remove the relationship between the metric and the size (*NOS*). The Figure 3.10 shows the relationship between 5 such randomly selected metrics and *NOS*, before and after normalization respectively. Only for *HBUG* and *MDN* metrics, we observe that the normalization has (almost) removed the relationship (see the fitted lines).

In the case of other three metrics, normalization was not able to remove the relationship; in fact it has reversed the direction of the relationship. This exercise demonstrates that dividing

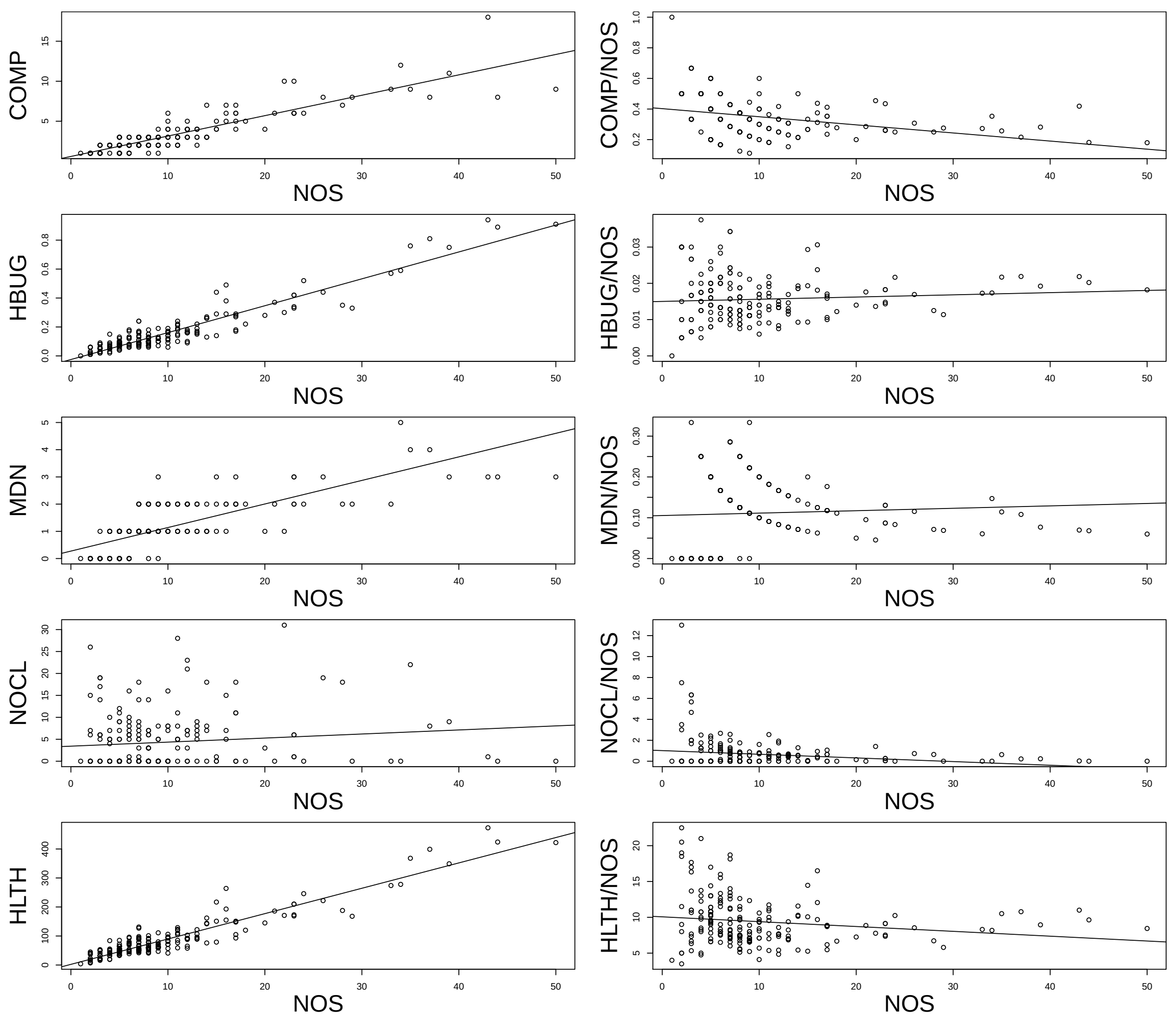


Figure 3.10: Scatter plots showing relationship between quality metrics on vertical-axis and size metric (*NOS*) on horizontal-axis, before and after carrying out normalization of metrics. The plots in the left column show the relationship between the metric and the size before normalization, whereas the plots on the right show the relationship between the normalized metrics and the size metric.

each metric with size does not always make the metrics independent of size. Therefore, we need another method to control for size.

3.1.6.2 Regression

We use Linear Regression Models (LRM) with two independent variables to control for the confounding effect of size as explained by Kutner et. al. in their book [115]. The two independent variables used are *NOS*, a quantitative variable quantifying the size of the methods and *isClone*, a binary qualitative variable indicating if a method is a cloned method (isClone =1) or not (isClone=0). For each project, we built LRMs for each of the metric (excluding *NOS* metric) as a response variable, giving us a total of $3,562 \times 26$ =92,612 LRMs. Out of these 92,612 LRMs we selected the ones where R-squared values were greater than or equal to 65%. This step ensured that we are using only those LRMs which could fit the data well enough to explain at least 65% of the variance in the given response variable. We further rejected those LRMs where the presence of *isClone* variable was not statistically significant ($p>0.05$) to explain the variance of the response variable. Moreover, we rejected the LRMs where the size of the effect (r), calculated using Hedge's g formula, was less then 0.1. In order to understand how LRMs are used to carry out size-control, consider the following example: In this example, the response variable is *XMET* metric, the two independent variables are *NOS* and *isClone* and the regression model is of the form:

$$E(XMET) = b_0 + b_1 \times NOS + b_2 \times isClone$$

where b_0, b_1, and b_2 are the parameters of the regression model. After fitting the regression model we get the following equation:

$$E(XMET) = 1.5510 + 0.42 \times NOS - 1.91 \times isClone$$

For the non-cloned methods ($isClone$=0), the equation becomes:

$E(\text{XMET}) = 1.55 + 0.42 \times NOS$

whereas for the cloned methods (*isClone*=1), the equation becomes:

$E(\text{XMET}) = -0.360251 + 0.42 \times NOS$

Figure 3.11 shows the fitted regression lines for cloned and non-cloned methods for the above example. Notice that both lines have the same slope (b_1=0.42), and therefore for any given level of *NOS*, the average distance between the two lines, given a fixed value of *NOS* is $b_2=-1.91$ units ($-$ sign indicating that cloned methods have lower XMET values than the non-cloned methods). Thus, -1.91 is the average difference between the cloned methods and non-cloned methods, when controlling for size. The R-squared value of the above model is 84% (>65%), the P-value of the *isClone* variable is 0.002 (<0.05) and the absolute effect size is 0.61 (>0.1). This model satisfies all our criteria and therefore we can use this model in our analysis.

The Figures 3.12, 3.13, and 3.14 show the results of the size control experiments using regression. For the *Complexity* (Figure 3.12) and *Documentation* (Figure 3.14) categories of metrics it becomes evident that the number of projects showing that cloned methods have lower metric values is almost equal to the number of projects where non-cloned methods have lower metric values. The metrics *NOA*(3), *CAST*(33), *NOCL*(23) have a very few number of projects and therefore we can safely ignore them from this analysis. The metrics *HEFF* and *NEXP* are the only two metrics where we see a clear difference between cloned and non-cloned methods. The cloned methods have won in *NEXP* metric, indicating that there are fewer number of expressions in the cloned methods, whereas the non-cloned methods have won in the *HEFF* metric, indicating that the effort to code non-cloned methods is lower than the effort required to code the cloned methods.

In the *Modularity* (Figure 3.13) category of metrics, it appears that the non-cloned methods have lower metric values (*MOD*(5), *LMET*(32), *EXCR*(7), and *EXCT*(7)) and therefore

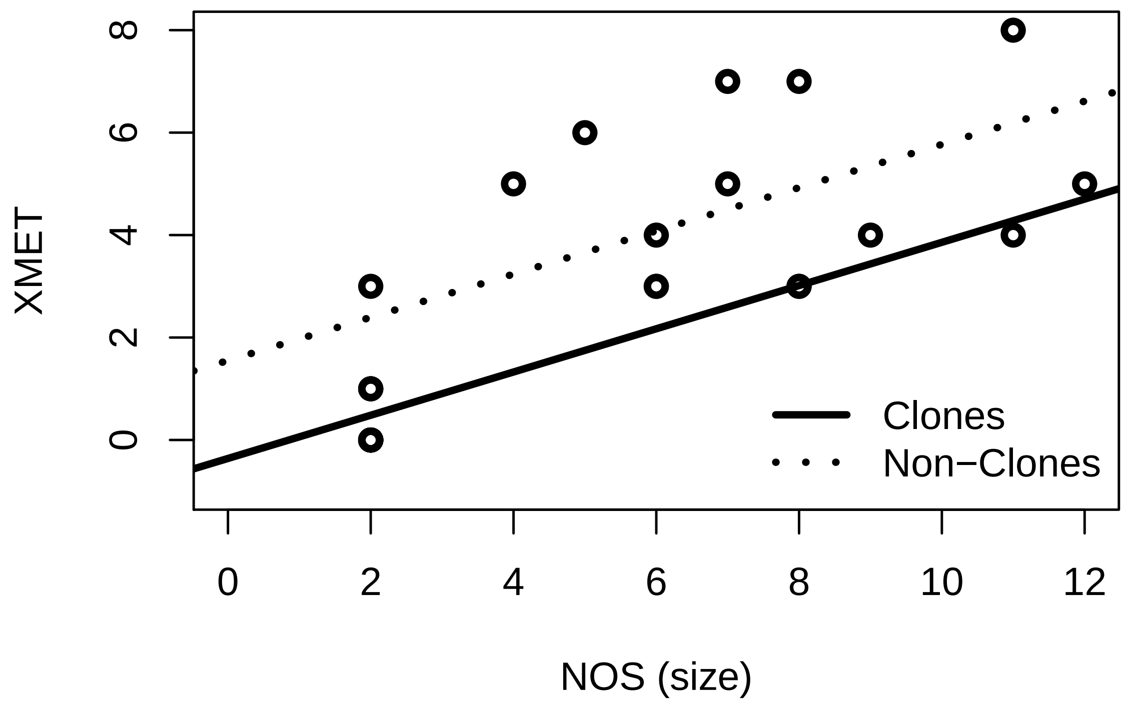


Figure 3.11: Fitted regression lines for cloned and non-cloned methods showing average difference between their XMET values. The vertical-axis shows the *XMET* metric values and the horizontal-axis shows the size of the methods as measured using *NOS* metric.

non-cloned methods appear to be more modular than the cloned methods. However, a very small number of projects (shown in the parenthesis) have shown these effects. For the *XMET* metric, around 70% of the 245 projects have shown that cloned methods have lower metric values, indicating that the cloned methods in these projects are more modular.

The results from this experiment are very different from the non-size-control experiment. We summarize the regression experiment results. *i) Only upto 18% of the projects from the population of 3,562 projects show statistically significant difference between the cloned and the non-cloned methods with at least a small effect for any of the quality metrics;* and *ii) The number of projects for which cloned methods have statistically significant (p<0.05, r>0.1) lower metric values is almost equal to the number of projects for which non-cloned methods have statistically significant lower metric values.*

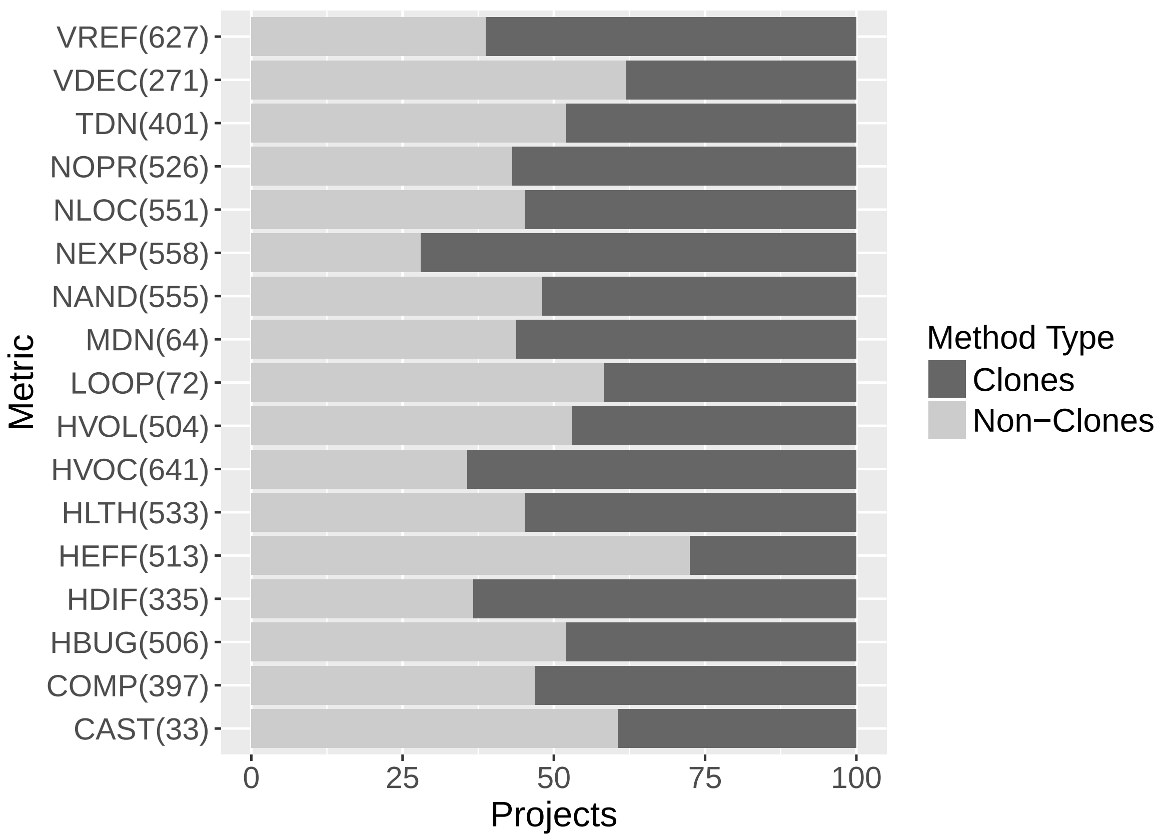


Figure 3.12: Regression: stacked bar chart comparing cloned and non-cloned methods.The Vertical-axis shows the *Complexity* metrics along with the number (in parenthesis) of subject projects that showed significant difference between cloned and non-cloned methods ($p<0.05$, $r>0.1$).

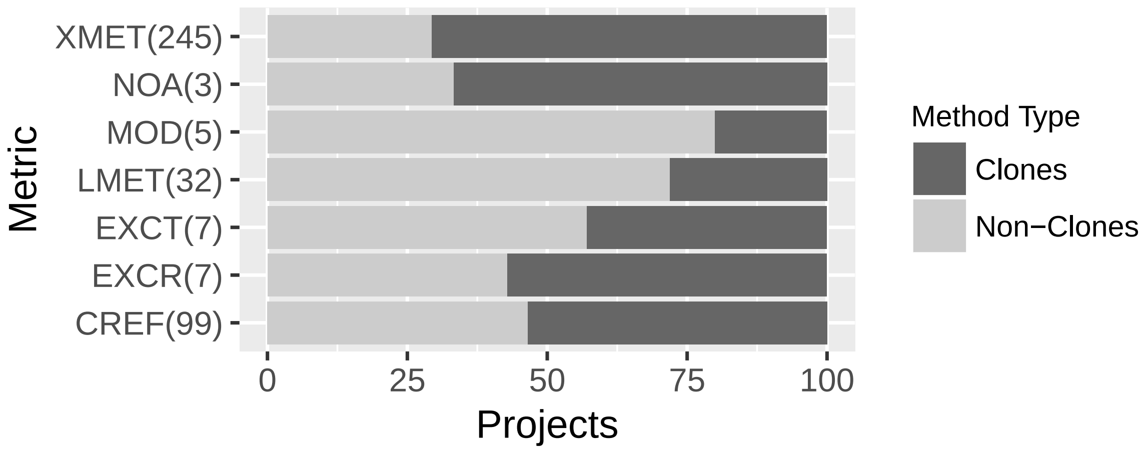


Figure 3.13: Regression: stacked bar chart comparing cloned and non-cloned methods.The Vertical-axis shows the *Modularity* metrics along with the number (in parenthesis) of subject projects that showed significant difference between cloned and non-cloned methods ($p<0.05$, $r>0.1$).

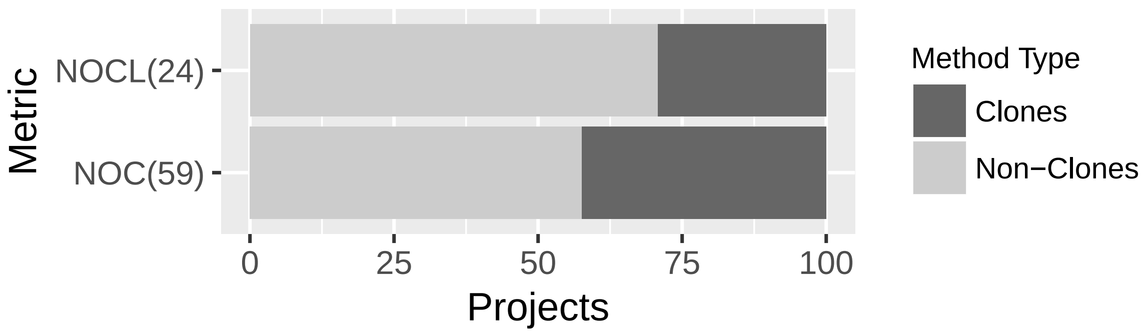


Figure 3.14: Regression: stacked bar chart comparing cloned and non-cloned methods.The Vertical-axis shows the *Documentation* metrics along with the number (in parenthesis) of subject projects that showed significant difference between cloned and non-cloned methods ($p<0.05$, $r>0.1$).

3.1.6.3 Methods without Method body

We noticed that there are some methods in our dataset which have no method-body (please refer Section 3.1.7.1 for details). There are 6,760 out of 412,705 cloned methods (1.6%) and 631 out of 616,604 non-cloned methods (0.1%) in our dataset with no method-body. We wanted to test if the presence of such methods could partially explain the size difference between cloned and non-cloned methods. To this end, we removed all such methods and their cloned methods from the dataset. After this, we rejected the projects that have less than or equal to 10 clone or 10 non-cloned methods, resulting into the rejection of 10 projects from the total of 3,562 projects. This new dataset consists of 405,903 cloned and 615,850 non-cloned methods in 3,562 projects.

We performed all three experiments – i) one without any size control, ii) one with size control where we divide each metric by the size metric- NOS, and iii) size control with Regression. Figures 3.15, 3.16, and 3.17 show results of the non-size-control experiments. Figures 3.18, 3.19, and 3.20 show the results of the size control experiments using *Dividing each metric by the size metric* method. Figures 3.21 3.22, and 3.23 show the results of the size control experiments using *Regression.*

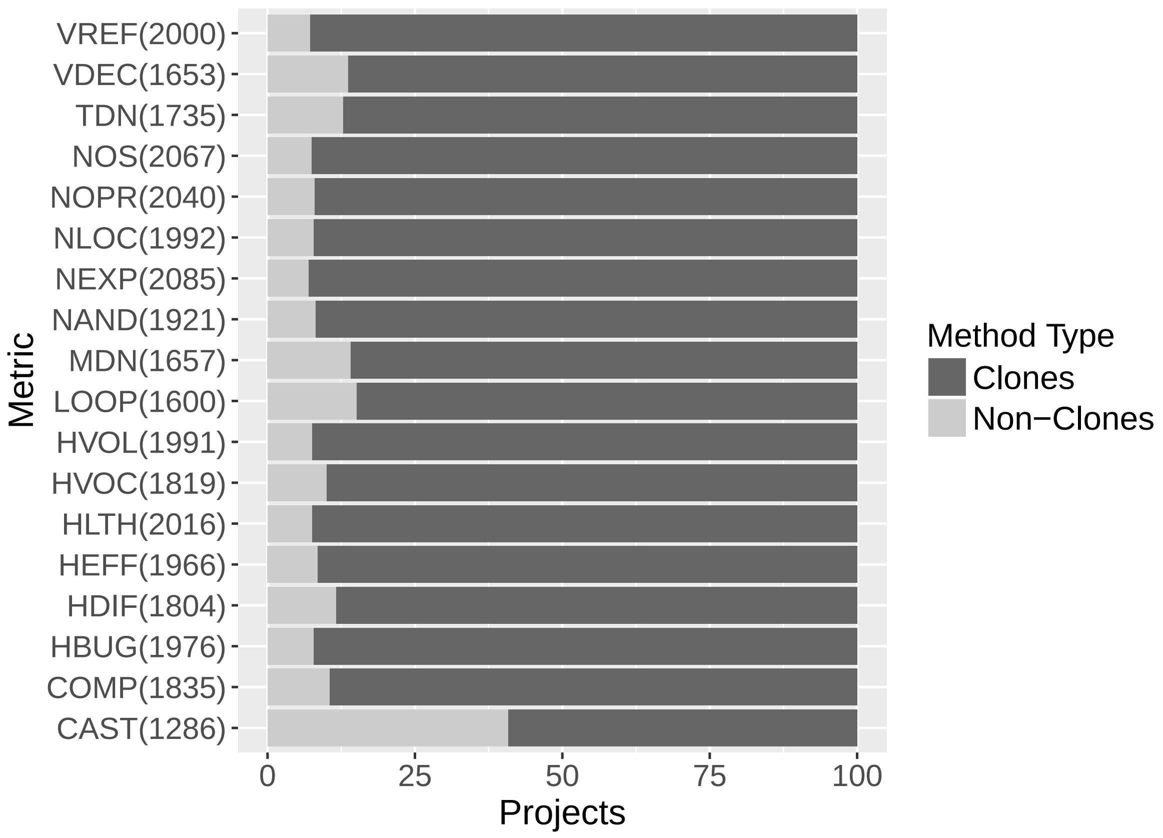


Figure 3.15: Wilcoxon Rank-Sum test: stacked bar chart comparing cloned and non-cloned methods.The Vertical-axis shows the *Complexity* metrics along with the number (in parenthesis) of subject projects that showed significant difference between cloned and non-cloned methods (p<0.05, r>0.1).

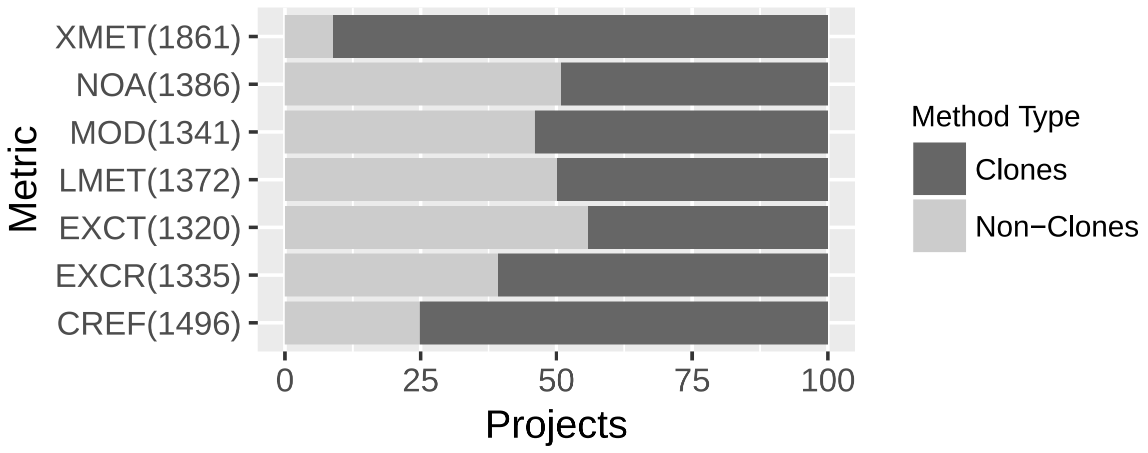


Figure 3.16: Wilcoxon Rank-Sum test: stacked bar chart comparing cloned and non-cloned methods.The Vertical-axis shows the *Modularity* metrics along with the number (in parenthesis) of subject projects that showed significant difference between cloned and non-cloned methods (p<0.05, r>0.1).

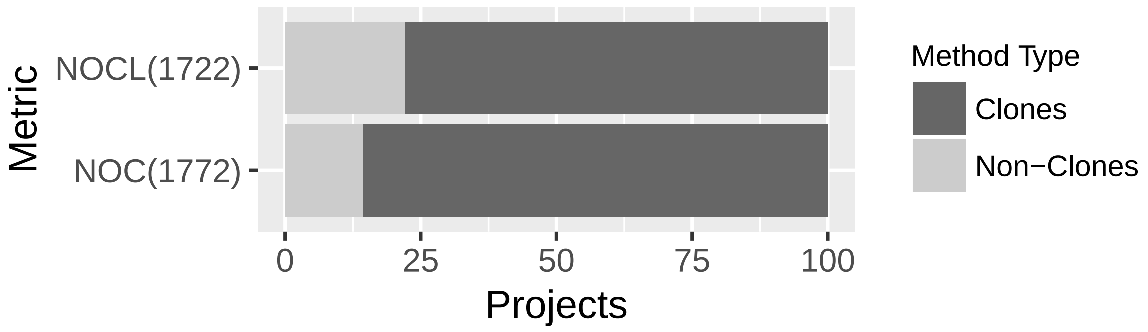


Figure 3.17: Wilcoxon Rank-Sum test: stacked bar chart comparing cloned and non-cloned methods.The Vertical-axis shows the *Documentation* metrics along with the number (in parenthesis) of subject projects that showed significant difference between cloned and no-cloned methods (p<0.05, r>0.1)

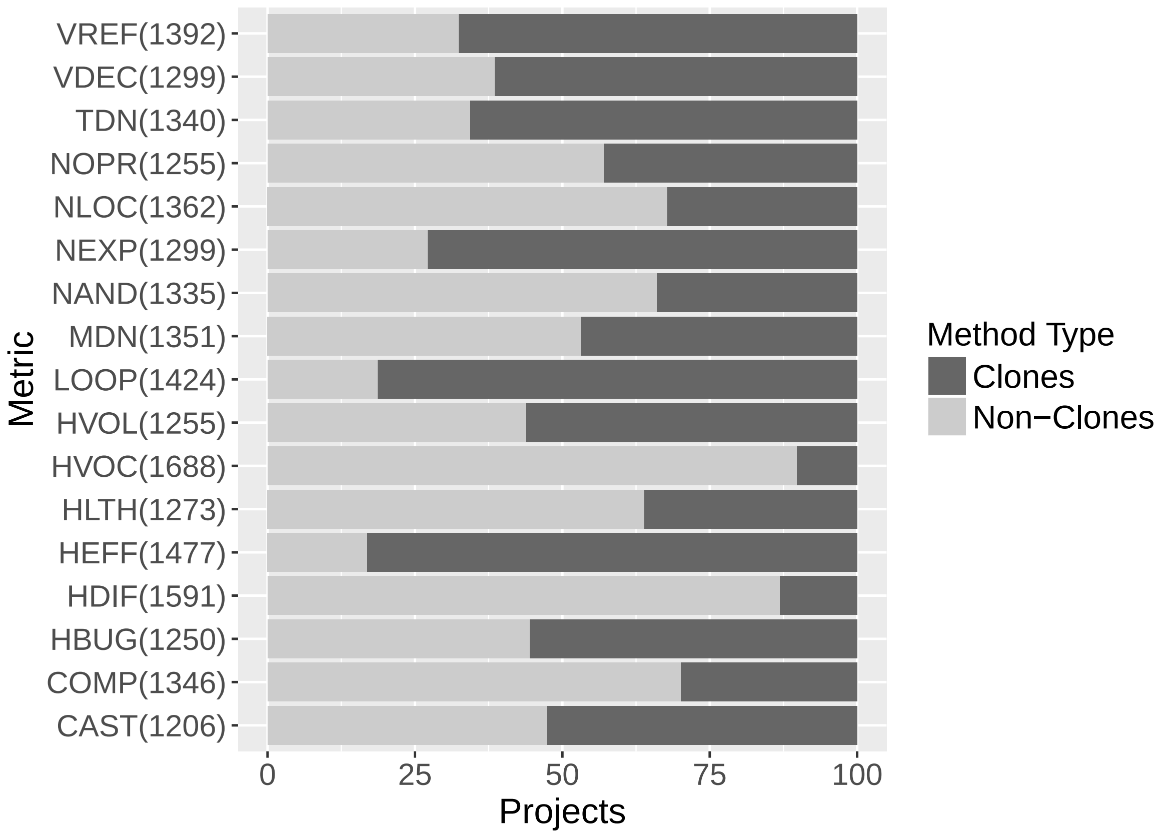


Figure 3.18: Wilcoxon Rank-Sum test: stacked bar chart comparing cloned and non-cloned methods.The Vertical-axis shows the normalized *Complexity* metrics along with the number (in parenthesis) of subject projects that showed significant difference between cloned and non-cloned methods (p<0.05, r>0.1).

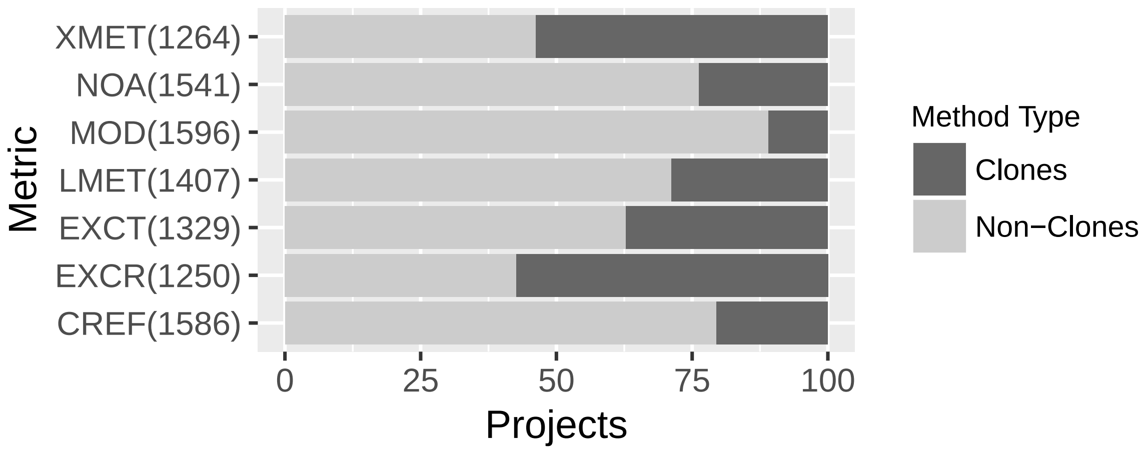


Figure 3.19: Wilcoxon Rank-Sum test: stacked bar chart comparing cloned and non-cloned methods.The Vertical-axis shows the normalized *Modularity* metrics along with the number (in parenthesis) of subject projects that showed significant difference between cloned and non-cloned methods (p<0.05, r>0.1).

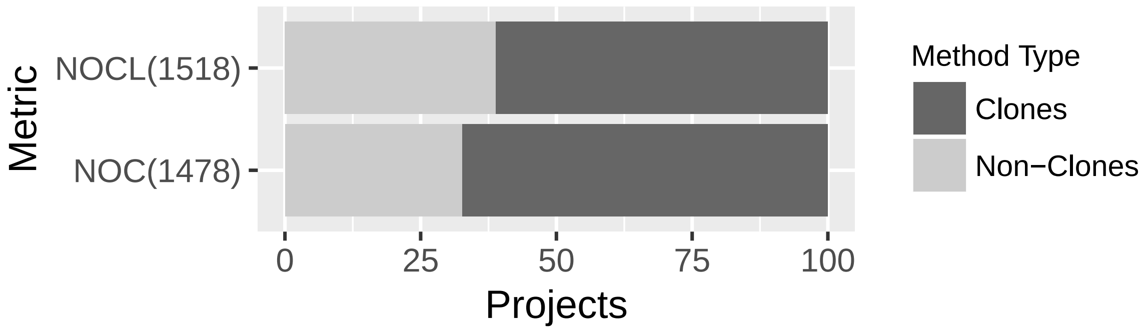


Figure 3.20: Wilcoxon Rank-Sum test: stacked bar chart comparing cloned and non-cloned methods.The Vertical-axis shows the normalized *Documentation* metrics along with the number (in parenthesis) of subject projects that showed significant difference between cloned and non-cloned methods (p<0.05, r>0.1).

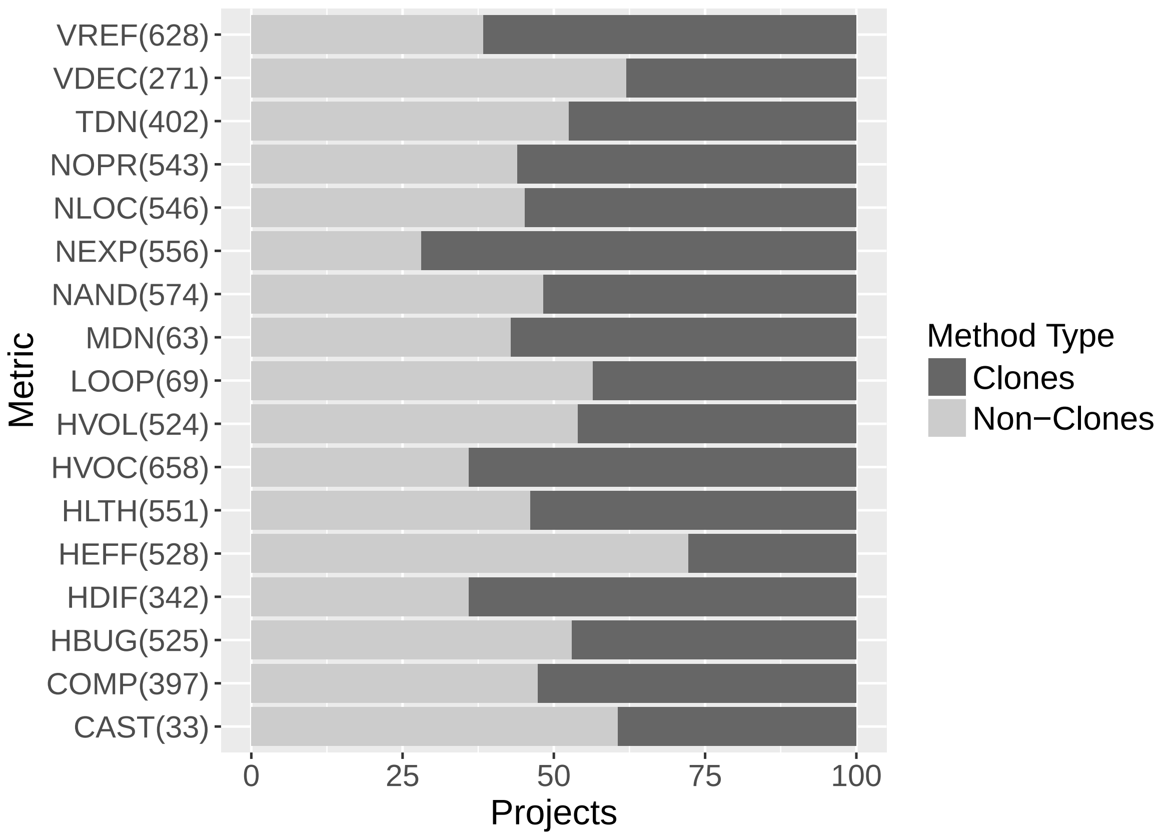


Figure 3.21: Regression: stacked bar chart comparing cloned and non-cloned methods.The Vertical-axis shows the *Complexity* metrics along with the number (in parenthesis) of subject projects that showed significant difference between cloned and non-cloned methods ($p<0.05$, $r>0.1$).

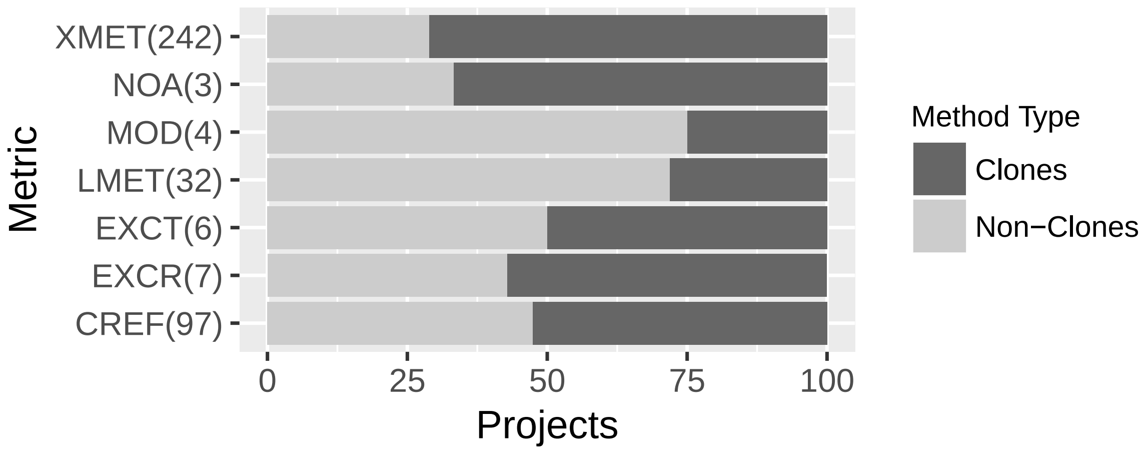


Figure 3.22: Regression: stacked bar chart comparing cloned and non-cloned methods.The Vertical-axis shows the *Modularity* metrics along with the number (in parenthesis) of subject projects that showed significant difference between cloned and non-cloned methods ($p<0.05$, $r>0.1$).

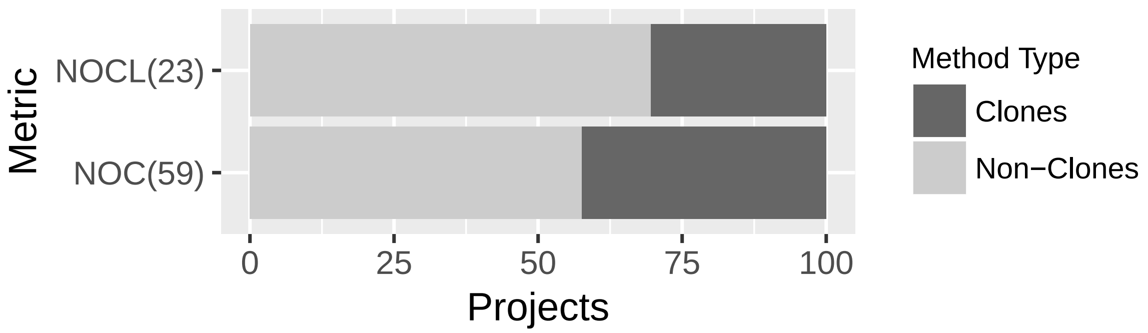


Figure 3.23: Regression: stacked bar chart comparing cloned and non-cloned methods.The Vertical-axis shows the *Documentation* metrics along with the number (in parenthesis) of subject projects that showed significant difference between cloned and non-cloned methods ($p<0.05$, $r>0.1$).

We observe similar results to what we had observed with the original dataset. Also, the clone methods are still around 18% smaller than the non-cloned methods. This exercise confirms that the presence of no method-body methods in the dataset does not explain why the cloned methods are on an average smaller than the non-cloned methods.

3.1.7 Mixed Method Analysis

The analysis we presented in the previous section points out that on an average there is a difference in the size of cloned and the non-cloned methods. In the experiments where we did not control for size, the cloned and non-cloned methods appear to be different with respect to the quality metrics. But, when we control for size (using Regression), the difference disappears. The analysis raised more questions: Why are cloned methods on an average smaller than the non-cloned methods? Is the size difference between the cloned and non-cloned method apparent to a human observer? Do human observers see any difference in the *comprehensibility* of cloned and non-cloned methods? Is there a difference in the *usefulness of the comments* present in the cloned and non-cloned methods? Further, we wanted to understand if cloned and non-cloned methods perform fundamentally different functionalities? and are they auto-generated or copy-pasted then modified? Are there any signs that could tell us that developers were aware of a method before they decided to create a clone of it?

In order to answer the above questions we conducted a mixed method analysis. This analysis was made in three parts. We first looked at a small subset of cloned and non-cloned methods in search for validation of the quantitative results, as well as patterns and correlations among different characteristics of the methods. On a second step, we looked at pairs of clone methods. Finally, we looked at groups of cloned methods. This section describes our analysis and findings.

Methodology. From the dataset we created two subsets, one with 150 randomly selected cloned methods and other with 150 randomly selected non-cloned methods. Each author, three in total, manually inspected 50 cloned and 50 non-cloned methods. The analysis was made along four dimensions, namely *Size*, *Comprehensibility*, *Usability of comments*, and *Functionality.* For each of the dimensions, the authors manually tagged their set of methods. One of the authors went through the entire 300 methods (150 cloned and 150 non-cloned) to cross-validate the tags. In the cases where there were conflicts, the authors resolved them by discussing the concern among each other followed by voting.

The conflicts occurred mostly in the *Comprehensibility* and the *Functionality* dimensions. For these two dimensions, we organized a calibration session to test the *Inter-rater reliability.* In the calibration session each author independently looked at a common subset of 10 cloned and 10 non-cloned methods and re-tagged them. In both, *Comprehensibility* and *Functionality* dimensions for about 1 in 4 (25%) cases we observed different judgments between any two authors. We note, for *Comprehensibility* dimension, in the cases of conflicts, the authors were off by 1 point on the Likert Scale. That is if one author scored a method as 4 (hard to comprehend) then the author, in the case of conflict, would have scored that same method either as 3 (medium hard) or 5 (very hard).

Test-rater reliability. Further, some time later (4 months), the authors, for the *Comprehensibility* dimension, re-tagged a random subset of 10 cloned and 10 non-cloned methods. These methods were selected from the authors' respective set of 50 cloned and 50 non-cloned methods. We did this exercise to see if the authors are consistent in their judgments. We observed that the authors were consistent in about 7 out of the 10 cases, but when they were off, they were mostly off by 1 point on the Likert scale. To understand the process, consider a case where an author has been asked to score the *Comprehensibility* of a set of methods, twice on a Likert scale. Once at time T1 and again at Time T2. The difference T2-T1 is about 4 months. The Likert scale uses options 1 to 5, where a score of 1 maps to the method being very easy to comprehend and a score of 5 maps to the method being very hard to understand. We observed that the author was mostly consistent, but when they were not, they were off by 1 point.

To address the inconsistency issue, we grouped the Likert options during the analysis. For example, in the *Comprehensibility* dimension, we grouped the scores of 1 (very easy) and 2 (easy) into 1' (easy). and the score of 4 (hard) and 5 (very hard) into 2' (hard). The score of 3 (medium hard) was removed from some parts of the analysis as it could fall into, owing to the off by 1 inconsistency, any of the original options 2 and 4.

The methodology presented here has elements of both qualitative and quantitative analysis, and hence it falls into the territory of the Mixed Methods Analysis [21].

3.1.7.1 Analysis of Methods

Size. We wanted to understand whether a method looks very small, medium, large, or very large to the observers. The Likert Scale uses options 1 to 5, where the size increases as we move from option 1 to option 5. Columns 2 to 6 in Table 3.3 correspond to each of these options. Column *Type* contains the type of the methods analyzed; column *VS*, which

Table 3.3: Size of cloned and non-cloned methods

Type	VS	S	M	L	VL
Clone	33	71	25	12	9
Non-Clone	30	54	42	10	14

corresponds to option 1 in the Likert scale, shows the number of the methods that are very small (3-8 *NLOC*); column *S*, (option 2) shows the number of methods that are small (9-20 *NLOC*); column *M* (option 3) contains the number of methods that are of medium size (21-40 *NLOC*); column *L* (option 4) shows the number of methods that are large (41-70 *NLOC*); and column *VL* (option 5) shows the number of methods that are very large in size (>70 *NLOC*). The above thresholds to categorize methods into different size options are selected using human intuition. The authors looked into a sample set of methods together and agreed upon the above thresholds.

For size, we observed a noticeable difference in the number of methods classified as small(S) and very small (*VS*), with cloned methods being more often in these two categories. Conversely, non-cloned methods were more often found to be very large (*VL*). To test whether the distribution of cloned and non-cloned methods across the size categories are statistically different, we conducted a Chi-Square test. The test result shows that the distributions are not statistically different. $\chi^2 = 8.0371, p = 0.09023, df = 4$, where χ^2 is the Chi-Square value, *df* is the degrees of freedom, p is the P-value.

Further, we wanted to see whether there is any difference between the distribution of cloned and non-cloned methods if we categorize the methods into two categories: i) methods which are very small or small and ii) methods which are large or very large. To this end, we performed another Chi-Square test after combining *VS* and *S* into one category, and *L* and *VL* into another category. Again, we did not observe any statistically significant difference. $\chi^2 = 0.77287, p = 0.3793, df = 1$

The result of this test seems to be different than what we had observed in the quantitative analysis. The median *NLOC* and median *NOS* for cloned methods are 13 and 9, whereas median *NLOC* and median *NOS* for non-cloned methods are 16 and 11. The difference in both of these size metrics for cloned and non-cloned methods is about 18%. But when the methods are categorized into different categories of size, the difference is not statistically significant. Therefore, we can not ignore the null hypothesis that the cloned and non-cloned methods, when classified into the broad size-categories as viewed by human observers, come from the same distributions.

Additionally, we also noticed that in three of the cloned methods in our samples there were no method bodies but only method signatures, meaning that these methods came from interfaces. However, they passed the threshold for analysis, because they have more than 25 tokens – these are methods with many parameters of complicated types. We note that we had removed all methods with less than 25 tokens to ignore empty methods, stubs, method contracts defined in interfaces, abstract classes and other interferences. The heuristic did remove many insignificant methods, but not all of them, as seen here. Listing 3.1 shows an example of one such method signature.

Listing 3.1: Example of an empty method found during the qualitative study

```
@Transactional(propagation = Propagation.SUPPORTS, readOnly = true)
public com.liferay.portal.model.PersistedModel getPersistedModel(
java.io.Serializable primaryKeyObj)
throws com.liferay.portal.kernel.exception.PortalException,
com.liferay.portal.kernel.exception.SystemException;
```

Upon further investigation, we found that for such methods JHawk calculates *NOS* as 1. There are 6,760 out of 412,705 cloned methods (1.6%) and 631 out of 616,604 non-cloned methods (0.1%) in our dataset with *NOS*=1. The presence of such methods could partially explain the size difference between cloned and non-cloned methods. To this end, we removed all such methods along with their clones and conducted the experiments explained in Section 3.1.6.3. We observe no change in the conclusions for the *Complexity*, *Modularity*, and *Documentation* metrics of cloned and non-cloned methods. Also, the size difference between

Type	VE	E	M	H	VH
Clone	31	65	28	13	13
Non-Clone	32	69	32	10	7

Table 3.4: Comprehensibility of cloned and non-cloned methods

the cloned and non-cloned methods still persists – cloned methods, on an average, are 18% smaller than the non-cloned methods with respect to the *NOS* metric.

Comprehensibility. We capture the comprehensibility of a method using a Likert Scale ranging from 1 to 5, where a score of 1 means that the method is very easy to understand and 5 indicates that the method is very difficult to understand. This is an entirely subjective judgment, which is based on many variables that are not controlled for, including our own experience. Nevertheless, we believe this exercise was worth doing, because it would allow us to detect patterns, if they were to exist. For example, would the cloned methods appear to be simpler than the non-cloned methods?

Comprehensibility here is loosely related to the complexity metrics of the previous section, but it is not the same. Code may be hard to understand even when the logic is very simple – e.g. when the variable names are incomprehensible or the reader is not familiar with the used APIs.

We note that the judges were not given any rubric to score the methods. They were allowed to create their own rubric. The judges independently found following criteria to score the methods: i) the methods with hard to understand identifiers (variables, types) are mostly harder to comprehend, ii) the methods with many conditional paths and nested conditions are harder to follow, iii) the methods which are bigger in size are usually harder to comprehend, as judges need more time and focus to comprehend such methods.

Table 3.4 shows the results. Column *Type* contains the type of the methods analyzed; column *VE*, which corresponds to option 1 in the Likert scale, shows the number of the methods that were very easy to understand; column E, (option 2) shows the number of methods that were easy to understand; column M (option 3) contains the number of methods that were neither easy nor hard to understand; column H (option 4) shows the number of methods that were hard to understand; and column VH (option 5) shows the number of methods that were very hard to understand.

The distributions of both types of methods are similar, indicating that there were no clear differences between cloned and non-cloned methods in terms of their comprehensibility. We note, however, that we classified more cloned methods in the VH category (13) than non-cloned methods (7) in that same category.

To test whether the distribution of cloned and non-cloned methods across the comprehensibility categories are statistically different, we conducted a Chi-Square test. The test result ($\chi^2 = 2.5932, p = 0.628, df = 4$) shows that the distributions are not statistically different.

Further, we wanted to see whether there is any difference between the distribution of cloned and non-cloned methods if we categorize the methods into two categories: i) methods which are very easy or easy to comprehend and ii) methods which are hard or very hard to comprehend. To this end, we performed another Chi-Square test after combining VE and E into one category, and H and VH into another category. Again, we did not observe any statistically significant difference ($\chi^2 = 1.5033, p = 0.220, df = 1$).

Listing 3.2 shows a small sized method which was classified as very easy to understand. The method uses familiar variable names and performs a task of calling *appendField* method. The method shown in Listing 3.3 is an example of a method classified as very hard to understand. The identifiers (method names, variable names, type names) used in the method do not help much in understanding their purpose. However, some identifiers like *Token*, *jj_scan_token*, *LBRACE*, and *RBRACE* do suggest that the method is probably part of some tokenization or parsing process.

Listing 3.2: Example of a very easy to understand method

```
public StringBuilder appendFields(ObjectLocator locator, StringBuilder buffer, ToStringStrategy
     strategy) {
    {
        List<Format> theFormat;
        theFormat = this.getFormat();
        strategy.appendField(locator, this, "format", buffer, theFormat);
    }
    {
        List<DCPType> theDCPType;
        theDCPType = this.getDCPType();
        strategy.appendField(locator, this, "dcpType", buffer, theDCPType);
    }
    return buffer;
}
```

Listing 3.3: Example of a very hard to understand method

```
private boolean jj_3R_92() {
   if (jj_scan_token(LBRACE)) return true;
   Token xsp;
   while (true) {
      xsp = jj_scanpos;
      if (jj_3R_121()) { jj_scanpos = xsp; break; }
   }
   if (jj_scan_token(RBRACE)) return true;
      return false;
   }
```

Not surprisingly, we observed some positive correlation between the size of the methods and their comprehensibility. Most of the methods that were hard to understand were also bigger in size. However, we found some methods that were small and yet hard to understand, such as the one shown in Listing 3.3.

Usability of Comments. We wanted to understand whether the provided comments are helpful in the comprehensibility of methods, and whether there were differences in the comments of cloned vs. non-cloned methods. As such, we captured the subjective usefulness

Type	NC	JC	UC	VUC
Clone	90	7	39	14
Non-Clone	76	3	49	22

Table 3.5: Usability of comments on cloned and non-cloned methods

of the comments using a Likert Scale between 1 and 4, where the usability of comments increases as we move from option 1 to option 4. This is related to the comments metrics of the quantitative analysis, but it's not the same. Specifically, the quantitative metrics cannot capture whether a comment is useful or not; only people can do that.

Columns 2 to 6 in Table 3.5 correspond to each of these options. Column *Type* contains the type of the methods analyzed; column *NC*, which corresponds to option 1 in the Likert scale, shows the number of methods which had no comments; column *JC*, (option 2) shows the number of methods which have mostly junk comments or commented out code or comments that were practically useless; column *UC* (option 3) contains the number of methods which have somewhat useful comments. These comments do help in the code comprehension or to understand the purpose of given methods; column *VUC* (option 4) shows the number of methods which have very useful comments. Again, judges were not given any rubric to score the methods. They used their own intuitions to score. We note that in the cases where the judges found a mix of junk and useful comments, priority was given to the useful comments.

The difference between the quality of comments is noticeable between cloned and non-cloned methods. There are 90 methods for which clones have no comments, whereas there are only 76 non-cloned methods with no comments. The presence of large number of auto-generated could possibly explain part of the difference as auto-generated methods usually have no comments. We also observe that the numbers of somewhat useful and extremely useful comments for non-cloned method are also larger than for the cloned methods.

To test whether the distribution of cloned and non-cloned methods across the usability of comments categories are statistically different, we conducted a Chi-Square test. The test result ($\chi^2 = 5.6949, p = 0.13, df = 3$) shows that the distributions are not statistically different.

Further, we wanted to see whether there is any difference between the distribution of cloned and non-cloned methods if we categorize the methods into two categories: i) methods with no or junk comments and ii) methods with useful or very useful comments. To this end, we performed another Chi-Square test after combining *NC* and *JC* into one category, and *UC* and *VUC* into another category. Again, we did not observe any statistically significant difference ($\chi^2 = 3.9727, p = 0.046, df = 1$).

When we looked into whether there is any correlation between the quality of comments and the size of the methods, we found no statistically significant correlation.

Listing 3.4 shows an example of very useful comments. The comments in this method not only help to understand what the code is supposed to do, but also help to understand the rationale behind the decisions made in the code.

Listing 3.4: A method with extremely useful comments

```
/**
* Execute the Mojo.
* <p>
* The metadata model is loaded, and the specified template is executed with
* any template output being written to the specified output file .
* </p>
*/
public void execute() throws MojoExecutionException
{
    try
    {
        if (modelIds == null)
        {
            modelIds = new ArrayList();
            modelIds.add(project.getArtifactId());
        }
// Load the metadata file from an xml file (presumably generated
// by an earlier execution of the build−metadata goal.
        Model model = IOUtils.loadModel(new File(buildDirectory, metadataFile));
// Flatten the model so that the template can access every property
// of each model item directly, even when the property is actually
// defined on an ancestor class or interface .
        new Flattener(model).flatten();
        generateConfigFromVelocity(model);
    }
    catch (IOException e)
    {
        throw new MojoExecutionException("Error during config generation", e);
    }
    catch (BuildException e)
    {
        throw new MojoExecutionException("Error during config generation", e);
    }
}
```

Listing 3.5 shows a method with *Javadoc* comments. These comments are practically useless for a developer who wishes to understand the code.

Listing 3.5: A method with junk comments

```
/**
* @param jsonValues
* @return
* @throws JSONException
*/
private static long[] convertJSONToLongArray(JSONArray jsonArray) throws JSONException
{
    long[] longArray = new long[jsonArray.length()];
    if (jsonArray != null && jsonArray.length() > 0)
    {
        for (int i = 0; i < jsonArray.length(); i++)
        {
            longArray[i] = jsonArray.getLong(i);
        }
    }
    return longArray;
}
```

Functionality. We wanted to understand whether cloned and non-cloned methods were doing fundamentally different things. As such, we analyzed these methods in terms of the functionality they seem to be implementing.

Coding. To label functionality with the methods, we use a standard method named *Coding*. Miles and Huberman note: *Codes are tags or labels for assigning units of meaning to the descriptive or inferential information compiled during a study. Codes are usually attached to 'chunks' of varying size — words, phrases, sentences or whole paragraphs* [107].

Coding Methodology. After a first pass, we tagged each method with our best guess of the functionality that they seem to implement. This exercise took around 5 hours per person to tag 100 methods each. After the tagging was done, we went through all the tags to create a set of normalized tags. To understand the normalization process, consider an example, a list containing the following non-normalized tags: *"Sorting"*, *"Sort"*, *"Parsing"*, *"Parser"*, *"Parse"*. To normalize this list, we replaced semantically similar tags with a common tag. After the normalization of tags, the methods were re-tagged with the normalized tags by replacing the original tag with their normalized counterpart.

We note that this process of tagging methods had some challenges. In many cases it was difficult to come up with an exclusive tag that fully captured the functionality of the method because of several factors – size of methods (bigger methods usually tend to do more than one thing, making it hard to associate them with one functionality); complex application specific logic; lack of context; among others. Also, there is no fixed set of functionalities that we could have used in tagging. Nevertheless, we believe this exercise was important, as it captured some of the essence of what we observed in the methods. The list of normalized tags along with their description is as follows.

- *AppLogic*, a method that usually performs multiple responsibilities, often manifested in the form of two or more method calls.
- *Wrapper*, a method that often validates certain pre-conditions, transforms the parameters (e.g. objects) to create parameters for another method call with a similar responsibility as the caller.
- *Iterator* a method that iterates over a collection and executes one or more statements in a loop.
- *Conditions*, a method that uses a series of if else or switch case conditions to do the main task.
- *Complex Setter*, a method that sets a value of a class attribute. It includes setter with some validation code, methods which set values of multiple fields, and also setters which set only one field. Listing 3.7 shows an example clone pair, where we tagged these methods

with *Complex Setter* tag. The methods set the value of a variable (*defaultPrefix* in the top, and *localPrefix* in the bottom snippet) to the one passed to their arguments. But before setting the variable, they obtain a write-lock, perform some sanity checks, and also raise an exception if the argument received is an illegal argument.

- *Complex Getter*, a method that returns a value of a class attribute or an object like a *getInstance* method found in singleton classes. It includes getters with some validation code, methods which execute one or more boolean expressions, and also getters which get only one field. Listing 3.8 shows an example clone pair, where we tagged these methods with *Complex Getter* tag. The methods return the requested object if the object is not null, else it creates an instance and returns it.

- *Parser*, a method that transforms a piece of code (e.g. String) based on grammar rules into another form (e.g. AST). Lexers are also included in this tag.

- *StringBuilder*, a method that builds a string or a byte array, usually by calling append many times. The methods tagged as StringBuilders need not use the *StringBuilder* and *StringBuffer* APIs. We used the term *StringBuilder* to broadly classify methods which reflect a similar behavior. Listing 3.6 shows an example clone pair, where we tagged these methods with *StringBuilder* tag. These methods, though, do not use Java's StringBuilder API, but their implementation is very similar to how one would use a *StringBuilder* API. They make multiple calls to the *append* method, appending characters to the *dest* object.

- *Math*, a method that takes numbers as input and performs a mathematical operation with them.

- *Pattern Matching*, a method that matches a pattern in a string.

- *Serializer*, a method that transforms an Object into string or bytes, or vice-versa.

- *Clone*, a method that clones another object.

- *Equals*, a method that overrides *equals* method.
- *HashCode*, a method that overrides *HashCode* method.
- *Testcase*, a method representing a unit test case to test a scenario.
- *Logger*, a method whose main responsibility is to log execution details.
- *FileIO*, a method whose main responsibility is to write/read to/from a file.
- *Socket*, a method which uses sockets to perform networking tasks.
- *NA*, methods which are usually overloaded with a series of very complex tasks and it is hard to classify them in any specific bucket of functionality.

Figure 3.24 shows the bar charts for cloned and non-cloned methods showing their distribution across functionality tags. We observe that many cloned and non-cloned methods are tagged with *AppLogic*, *Wrapper*, *Iterator*, *Conditions*, *Getter*, *Setter*, *Parser*, and *StringBuilders*. In most of these tags, except for *Wrapper*, *Parser*, and *StringBuilder*, there are more non-cloned methods than cloned methods. The observation indicates that although cloning exist in most of the functionalities, there exists some functionalities where methods have many structural similarities to be deemed clones.

To test whether the distribution of cloned and non-cloned methods across the functionality tags are statistically different, we conducted a Chi-Square test. The test result ($\chi^2 = 35.119, p = 0.009, df = 18$) shows that the distributions are statistically different.

Listing 3.6: An example clone pair (Type-3), demostrating string builder like behavior.

```
    private static void appendTextRow( final ByteBuffer buffer , final Appendable dest, final int startPos,
            final int columns) throws IOException {
        final int limit = buffer.limit();
        int pos = startPos;
        dest.append('[');
        dest.append(' ');
        for (int c = 0; c < columns; c ++) {
            for (int i = 0; i < 8; i ++) {
                if (pos >= limit) {
                    dest.append(' ');
                } else {
                    final char v = (char) (buffer.get(pos++) & 0xff);
                    if (Character.isISOControl(v)) {
                        dest.append('.');
                    } else {
                        dest.append(v);
                    }
                }
            }
            dest.append(' ');
        }
        dest.append(']');
    }
```

```
    private static void appendHexRow( final CharBuffer buffer, final Appendable dest, final int startPos,
            final int columns) throws IOException {
        final int limit = buffer.limit();
        int pos = startPos;
        for (int c = 0; c < columns; c ++) {
            for (int i = 0; i < 8; i ++) {
                if (pos >= limit) {
                    dest.append(' ');
                } else {
                    final char v = buffer.get(pos++);
                    final String hexVal = Integer.toString(v, 16);
                    dest.append("0000".substring(hexVal.length()));
                    dest.append(hexVal);
                }
                dest.append(' ');
            }
            dest.append(' ');
            dest.append(' ');
        }
    }
```

Listing 3.7: An example clone pair (Type-3), demostrating complex setter behavior.

```
public void setDefaultPrefix( final String prefixString ) {
    try {
        lock.writeLock().lock();
        if (prefixString == null || StringUtils.isBlank(prefixString)) {
            defaultPrefix = null;
        } else {
            if (prefixes != null) {
                if (!prefixes.contains(prefixString)) {
                    throw new IllegalArgumentException("The given prefix is not part of the
                        prefixes.");
                }
            }
            defaultPrefix = prefixString;
        }
    } finally {
        lock.writeLock().unlock();
    }
}
```

```
public void setPrefix( final String prefixString ) {
    try {
        lock.writeLock().lock();
        if (prefixString == null || StringUtils.isBlank(prefixString)) {
            localPrefix.remove();
        } else {
            if (prefixes != null) {
                if (!prefixes.contains(prefixString)) {
                    throw new IllegalArgumentException("The given prefix is not part of the
                        prefixes.");
                }
            }
            localPrefix.set(prefixString);
        }
    } finally {
        lock.writeLock().unlock();
    }
}
```

Listing 3.8: An example clone pair (Type-2), demostrating complex getter behavior.

```
public HTML getOptionsCaption() {
    if (optionsCaption == null) {
        optionsCaption = new HTML();
        optionsCaption.setStyleName(CLASSNAME + "-caption-left ");
        optionsCaption.getElement().getStyle()
        .setFloat(com.google.gwt.dom.client.Style.Float.LEFT );
        captionWrapper.add(optionsCaption );
    }
    return optionsCaption ;
}
```

```
public HTML getSelectionsCaption()  {
    if (selectionsCaption == null) {
        selectionsCaption = new HTML();
        selectionsCaption.setStyleName(CLASSNAME + "-caption-right ");
        selectionsCaption.getElement().getStyle()
        .setFloat(com.google.gwt.dom.client.Style.Float.RIGHT );
        captionWrapper.add(selectionsCaption );
    }
    return selectionsCaption;
}
```

3.1.7.2 Analysis of Clone Pairs

After looking at cloned and non-cloned methods individually, we wanted to find out more about the clones, and why they were being identified as such. Are they auto-generated or copy-pasted then modified? Are there any signs that could tell us that developers were aware of a method before they decided to create a clone of it? This required us to do a deeper analysis of clone methods where we not only looked at the code of the clone pairs, but also the Java files they come from. To this end, we analyzed 32 randomly selected clone pairs in search for the type of cloning among the pairs of methods. The number 32 was selected arbitrarily.

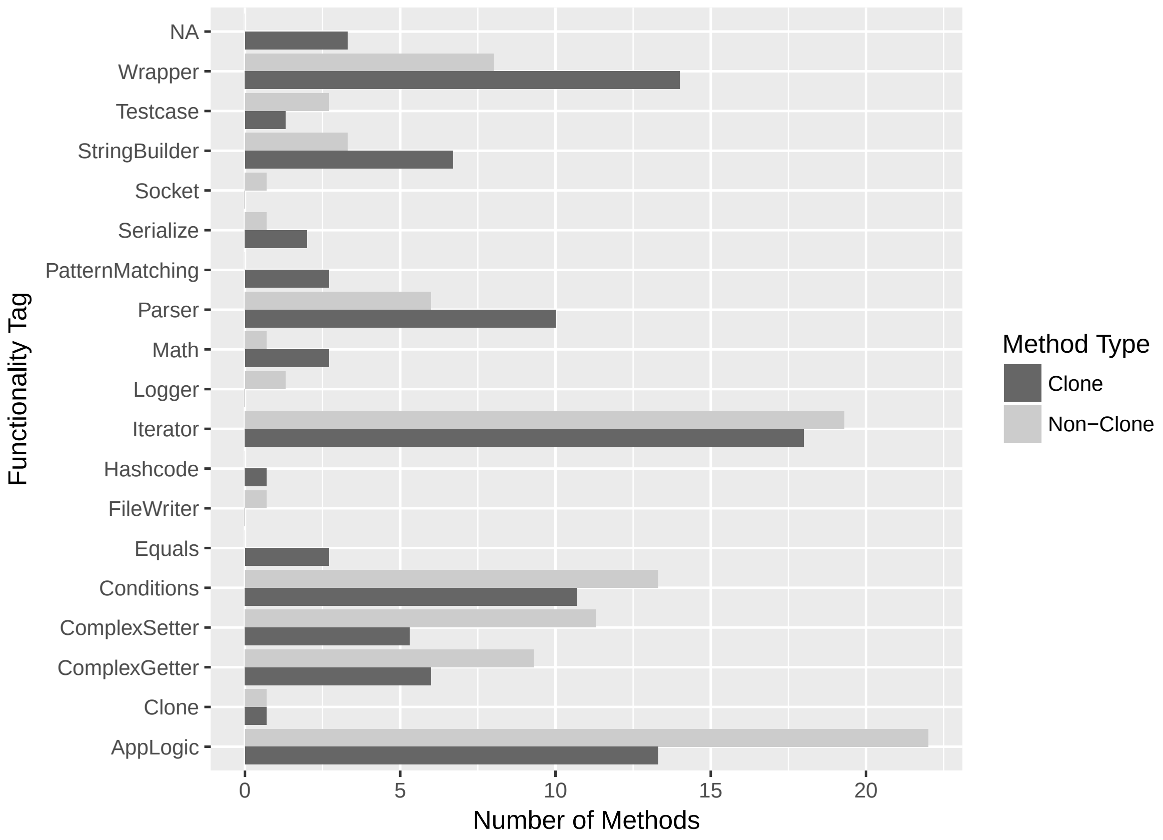


Figure 3.24: Distribution of cloned and non-cloned methods across functionality tags. The Vertical-axis shows the functionality tags and the Horizontal-axis shows the number of methods in each functionality tag.

Type-1	Type-2	Type-3	Same File	A.Gen.
4	16	12	13	5

Table 3.6: Results: analysis of 32 clone pairs

Table 3.6 shows the results of this analysis. The table has two sides, to distinguish between the type of cloning (1, 2 or 3) and the two additional characteristics we found interesting: whether the methods were found in the same file or not, and whether they were automatically generated or not.

As expected we found a variety of types of cloning, from identical (Type-1) to less identical (Type-3). There were 13 instances where both members of a clone pair have the same Java source file. We looked into the source file for each clone pair and found that in 7 out of 13 cases these methods appear consecutively next to each other. Also, none of the 13 methods appear to be auto-generated. We infer from the above two observations that the developers were possibly aware of the existence of these 7 similar methods. However, a more in-depth analysis of version history of these files would be required to conclude with confidence that the developers were aware of these similar methods.

In all of these cases of same-file clones, separation of concerns seemed to be the design rationale that the developers chose to follow. For example, consider the clone pairs shown in Listing 3.9. The method on the top, *expandItemsRecursively*, is structurally very similar to the method at the bottom, *collapseItemsRecursively*. The *expandItemsRecursively* method calls *expandItem* inside the *while* loop, whereas the other method calls *collapseItem* method inside the *while* loop. Also, there is an additional call to *requestRepaint* method made inside the *expandItemsRecursively* method. Functionally these two methods are performing the opposite functions, but structurally these are Type-3 clones. The developers chose to implement the two functions separately in two different methods instead of abstracting the commonalities into a third method.

Listing 3.9: Type-3 clones, demonstrating separation of concerns

```
public boolean expandItemsRecursively(Object startItemId){
    boolean result = true;
    // Initial stack
    final Stack<Object> todo = new Stack<Object>();
    todo.add(startItemId);
    // Expands recursively
    while (!todo.isEmpty()){
        final Object id = todo.pop();
        if (areChildrenAllowed(id) && ! expandItem(id, false) ){
            result = false;
        }
        if (hasChildren(id)){
            todo.addAll(getChildren(id));
        }
    }
    requestRepaint();
    return result;
}
```

```
public boolean collapseItemsRecursively(Object startItemId){
    boolean result = true;
    // Initial stack
    final Stack<Object> todo = new Stack<Object>();
    todo.add(startItemId);
    // Collapse recursively
    while (!todo.isEmpty()) {
        final Object id = todo.pop();
        if (areChildrenAllowed(id) && ! collapseItem(id) ){
            result = false;
        }
        if (hasChildren(id)) {
            todo.addAll(getChildren(id));
        }
    }
    return result;
}
```

We observed 5 instances where the methods in the clone pairs were generated using some code generator. In all such instances the members of the clone pair were found in different

Java source files. All of these clone pairs were Type-2 clones. Listing 3.11 shows a clone pair that was generated using JavaTM Architecture for XML Binding (JAXB). Both methods look very similar in structure and functionality. Each method copies an object of a different type to another object. We looked at the source files for these two methods; the method on the top comes from a file named *FeatureListURL.java* and the method on the bottom comes from a file named *StyleSheetURL.java*. We found comments in both of these files, shown in Listing 3.10, indicating that both these files were auto-generated.

Listing 3.10: commnets taken from StyleSheetURL.java indicating that the source file are auto-generated

```
//
// This file was generated by the JavaTM Architecture for XML Binding(JAXB) Reference
    Implementation , vhudson−jaxb−ri−2.1−2
// See <a href="http://java.sun.com/xml/jaxb">http://java.sun.com/xml/jaxb</a>
// Any modifications to this file will be lost upon recompilation of the source schema.
// Generated on: 2011.02.20 at 12:49:02 PM MEZ
//
```

One could write a generic method to avoid creation of clones, but since these are generated methods, maintainability may not be important. Should there be any need of bug fixes or modifications, the developers can modify the code generator and all members of the clone groups will get updated.

Listing 3.11: Type-2 clones, demonstrating generated methods

```
public Object copyTo(ObjectLocator locator, Object target, CopyStrategy strategy) {
    final Object draftCopy = ((target == null)?createNewInstance():target);
        if (draftCopy instanceof FeatureListURL) {
            final FeatureListURL copy = ((FeatureListURL) draftCopy);
            if (this.format!= null) {
                String sourceFormat;
                sourceFormat = this.getFormat();
                String copyFormat = ((String) strategy.copy(LocatorUtils.property(locator, "format",
                        sourceFormat), sourceFormat));
                copy.setFormat(copyFormat);
            } else {
                copy.format = null;
            }
            if (this.value!= null) {
                String sourcevalue;
                sourcevalue = this.getvalue();
                String copyvalue = ((String) strategy.copy(LocatorUtils.property(locator, "value",
                        sourcevalue), sourcevalue));
                copy.setvalue(copyvalue);
            } else {
                copy.value = null;
            }
        }
        return draftCopy;
    }
```

```
    public Object copyTo(ObjectLocator locator, Object target, CopyStrategy strategy) {
        final Object draftCopy = ((target == null)?createNewInstance():target);
        if (draftCopy instanceof StyleSheetURL) {
            final StyleSheetURL copy = ((StyleSheetURL) draftCopy);
            if (this.format!= null) {
                String sourceFormat;
                sourceFormat = this.getFormat();
                String copyFormat = ((String) strategy.copy(LocatorUtils.property(locator, "format",
                        sourceFormat), sourceFormat));
                copy.setFormat(copyFormat);
            } else {
                copy.format = null;
            }
            if (this.value!= null) {
                String sourcevalue;
                sourcevalue = this.getvalue();
                String copyvalue = ((String) strategy.copy(LocatorUtils.property(locator,"value", sourcevalue),
                        sourcevalue));
                copy.setvalue(copyvalue);
            } else {
                copy.value = null;
            }
        }
        return draftCopy;
    }
```

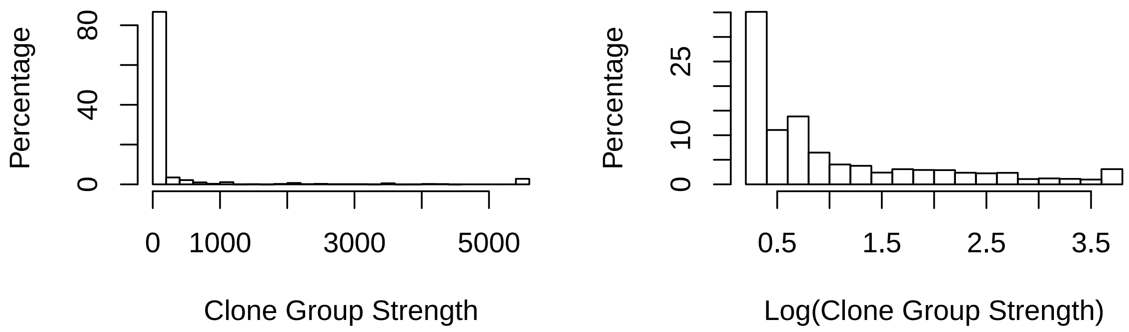


Figure 3.25: Distribution of clone groups. Left: histogram where number of members in a clone group are shown in linear scale(X-axis). Right:histogram where number of members in a clone group are shown in $\log_{10}$ scale(X-axis)

3.1.7.3 Analysis of Clone Groups

In order to understand why there are more small methods, we turned to analyzing clone groups, i.e. the set of methods that were identified as clones of a common method. To this end, we divided the cloned methods from subsection 3.1.7.1 into two subsets: i) set of cloned methods which are very small (VS and small S); and ii) the set of cloned methods which are large (L and very large VL). For each method in these two sets, we created a set of methods that were reported as its clones. This gave us the clone groups for each method. We named the clone groups obtained from smaller (VS and S) methods as *Clone Groups with Small Methods*. Conversely, we named the clone groups obtained from the larger (L and VL) as *Clone Groups with Large Methods.* We then analyzed these groups looking for differences and similarities in these groups.

Figure 3.25 shows the distribution of clone groups in the entire dataset. The X-axis in the left histogram represents the number of members in the clone groups. Whereas the X-axis in the right histogram has been transformed into the $\log_{10}$ scale. The Y-axis, in both histograms, represents the percentage of clone groups.

To get a better picture of the distribution, we are showing the histogram in linear and log scale. We call clone groups with 2 to 10 members *Small clone groups*, clone groups with

11 to 100 members as *Medium clone groups*, clone groups with 101 to 1000 as *Large clone groups*, and clone groups with more than 1000 members are *Very large clone groups.* The authors decided and agreed upon these thresholds based on human intuitions.

There are approximately 60% of the clone groups that fall into *Small clone group* ($\log_{10} 0.5 = 3.16$,$\log_{10} 1.5 = 31.6$, $\log_{10} 2.5 = 316.22$, and $\log_{10} 3.5 = 3162.2$); around 20% in *Medium clone group*; and around 20% in *Large* and *Very large clone groups* combined. We note that these clone groups should not be confused with clone classes. In a clone class, all members of the class are reported to be clones of each other, whereas in a clone group, all members of the group are reported to be clones of one method. It could be possible that some or all of the members of a clone group could be clones of each other, but it is not a necessary condition. We did not compute clone classes because computing clone classes is not trivial and SourcererCC currently does not support reporting clone classes.

Clone Groups with Small Methods. While looking at the clone groups we saw some *large clone groups* (>100 methods in a group), hinting towards the use of code generators. We looked at the Java source files for some of the methods, specifically looking for comments indicating the use of code generators; we found many instances where the methods were auto-generated. We noticed some patterns: i) the large clone groups usually indicate the use of a code generator; ii) the cloned methods which are auto-generated usually exhibit Type-2 cloning, however we did find some instances of Type-3 clones as well; and iii) smaller auto-generated methods show more instances of Type-2 cloning than Type-3, which are more prevalent in larger methods.

Table 3.7 shows some of the largest clone groups that we observed. Column *Project* shows the name of the project where the clone group was found; column *Count* shows the number of methods present in the group; column *Type* shows the type of clones we observed (Type-1, Type-2, and Type-3); the last column, *A.Gen.* shows whether the methods in the clone group are auto-generated or not; the Value *?*, means that we are not sure, in the absence of

Project	Count	Type	A.Gen.
jython-standalone@2.5.2	5,474	2	Yes
hapi-osgi-base@2.0-beta1	4,421	2,3	Yes
geronimo-schema-jee_5@1.2	2,974	2	Yes
geronimo-schema-javaee_6@1.0	2,288	2	Yes
cxf-testutils@2.6.0	2,077	2	Yes
portal-service@6.1.0	1,125	2,3	?
cmlxom@3.1	880	2	Yes
openejb-itests-client@4.0.0-beta-2	575	2	?
aws-java-sdk@1.3.8	378	2	Yes
dsl-pom@5.0.0	126	2	Yes
trove4j@3.0.2	126	2	?
axis2-adb@1.6.2	94	1	?

Table 3.7: Clone groups of smaller (VS and S) methods

any reliable indicators, if the members were auto-generated.

As the table shows, most of the large clone groups that we saw were auto-generated, which, in retrospect, is not surprising. Row 1 in Table 3.7 shows one large clone group with $5,474$ members. This clone group belongs to jython-standalone@2.5.2 project. We looked at around 100 random methods of this clone group and they all were auto-generated and showed Type-2 cloning. Listing 3.12 shows an example pair for this clone group.

Listing 3.12: An example clone pair from jython-standalone@2.5.5 project

```
public PyObject __add__(PyObject other) {
    PyType self_type=getType();
    PyObject impl=self_type.lookup("__add__");
    if (impl!=null) {
        PyObject res=impl.__get__(this, self_type). __call__ (other);
        if (res==Py.NotImplemented)
            return null;
        return res;
    }
    return super.__add__(other);
}
```

```
public PyObject __radd__(PyObject other) {
    PyType self_type=getType();
    PyObject impl=self_type.lookup("__radd__");
    if (impl!=null) {
        PyObject res=impl.__get__(this, self_type). __call__ (other);
        if (res==Py.NotImplemented)
            return null;
        return res;
    }
    return super.__radd__(other);
}
```

Similarly, in other large clone groups (Table 3.7), we observed that code generated using templates is one of the major causes behind the structural similarities in the members of large clone groups.

Clone Groups with Large Methods. Table 3.8 shows some of the clone groups that we observed in the set of large (L and very large VL) cloned methods. Structurally, the table is identical to Table 3.7.

We observed a pattern similar to what we observed in Section 3.1.7.3 – the larger clone groups contain methods that are auto-generated. Row 1 in Table 3.8 shows a large clone group (2,345 members). The members of this group are auto-generated using *JacORB IDL compiler*, as mentioned in their Java source files. Similarly, we found other clone groups where

Project	Count	Type	A.Gen.
jacorb@2.3.1	2,345	2,3	Yes
iso-19139-d_2006_05_04-schema@1.0.3	209	2,3	Yes
sfee@1.0.4	165	2,3	Yes
esper@4.6.0	21	2	Yes
jackrabbit-spi-commons@2.4.1	17	2,3	Yes
portal-impl@6.1.0	4	3	No

Table 3.8: Clone groups of larger (*L* and *VL*) methods

the cloning can be associated to code generators. Unlike our observations in Section 3.1.7.3, where Type-2 dominated the clones in larger clone groups, we saw both, Type-2 and Type-3 clones as shown in Table 3.8.

Small vs. Large Cloned Methods. Comparing tables 3.7 and 3.8, it appears that methods which are automatically generated using some template, and that end up being very similar, tend to be small rather than large. Table 3.7, with the small methods, shows many very large clone groups with more than 1,000 methods, whereas Table 3.8 shows only one clone group with more than 1,000 methods.

3.1.7.4 Summary of Mixed Method Analysis Findings

To understand why cloned methods, on average, were found to be smaller than the non-cloned methods we performed the mixed method analysis. We also compared the two types of methods based on their *comprehensibility* and *usefulness of comments* found in them. Additionally we looked into whether cloned and non-cloned methods perform similar functionalities or not.

Our analysis allowed us to gain some additional insights as to *why* cloned methods in our dataset are smaller than non-cloned methods. We found one possible factor that may lead to the size differences – among the methods identified as clones, a very large number of them seem to be automatically generated. We found many very large clone groups with more than

1,000 methods each. Furthermore, it seems that these automatically generated methods tend to be smaller rather than larger. This may explain why so many cloned methods are small, rather than following the size distribution of their non-cloned counterparts.

The analysis pointed out to the presence of bodiless methods (*interfaces*) in our dataset. This could have impacted our findings; so we performed the quantitative analysis (see Section 3.1.6.3) again after removing such methods and their clones from the dataset. The results did not change our findings; ensuring that the number of such bodiless methods were not enough to impact the findings of this study.

We did not observe any statistically significant difference between cloned and non-cloned methods with respect to the *comprehensibility* and *usability of comments*.

We also found that certain functionality (e.g. parsing and building strings) tends to be more pervasive in cloned methods than non-cloned methods, suggesting that these kinds of functionality often come in multiple, but similar, flavors for the same objects.

Additionally, we gained some more insights about cloned methods. Specifically, among the clones that do not seem to be automatically generated, we found many instances of multi-purpose methods, i.e. methods that have very similar structure and vocabulary but do the opposite (expand vs. collapse, etc.).

3.1.8 Limitations of this Study

The results of this study are from a large number of open source software projects written in Java. Although we chose subject projects that exhibit variety in their size and type, several factors such as development practices within OSS communities, release cadence, and team size might have an impact on the observed phenomena. Thus drawing general conclusions from empirical studies such as this is difficult because the results depend on many relevant confounding variables [13]. For this reason, we cannot assume a priori that the results of this study generalize beyond the setting for which it was conducted. However, the overall results of this study showed several commonalities across a wide range of OSS Java projects and indicate that the results hold for more than just the studied projects.

Moreover, the clone detection studies are affected by the configuration of the tools [157]. We mitigate this risk by using the configuration of SourcererCC for which the tool authors have reported a very high precision and recall. SourcererCC has been previously compared to other state of the art clone detectors and shown to deliver state of the art precision and recall [137]. Also, we conducted a manual validation of a statistically significant sample of SourcererCC's output. The validation suggests that the SourcererCC is accurate on the dataset used in this study, giving 97% precision.

A major part of this study is to generate the dataset by joining results from two different tools. We used fully qualified method name (FQMN) as a common field to join the results. The join merges the two results, but produces inconsistent output in the presence of overloaded methods, owing to the same FQMNs. We addressed this issue by removing all overloaded methods from our dataset. We note that the removal of overloaded methods may introduce a systematic bias, as one could argue that overloaded methods are usually similar in their semantics which could lead to similar code. Though this step removed a lot of methods from our dataset, our dataset still contains a large number of cloned and non-cloned methods to instill confidence in the results.

In the absence of reliable indicators and tools to automatically identify auto-generated code from the manually written code, we could not test if and how much of the difference in the size of cloned and non-cloned methods could be explained by the presence of auto-generated code. The random samples that we manually analyzed do indicate that the auto-generated methods could be one possible factor that leads to the size difference.

Another concern is the reliability in the mixed method analysis. That is, if the analysis is replicated, would the findings be the same? To this end, we conducted a calibration session where we answered two questions: i) how often do the judges agree with each other, and ii) what is the judges' consistency in making a choice on the Likert scale. We found that the judges agree, on an average, with each other 4 out of 5 times and that they are consistent on the Likert scale 7 out of 10 times. But when they are inconsistent, they are inconsistent mostly by 1 point on the Likert scale.

3.1.9 Conclusion

We conducted a statistical study to explore if there exists any difference between the quality of cloned methods and non-cloned methods, as measured by well-known software quality metrics. The dataset consists of 3,562 open source Java projects containing 412,705 cloned and 616,604 non-cloned methods. The study uses 27 software metrics, as a proxy for quality, spanning across complexity, modularity, and documentation categories.

Statistically we find no evidence that the cloned methods are different from the non-cloned methods for most of the metrics. The main exception pertained to size: we found that cloned methods, on an average (median *NOS*), are 18% smaller than the non-cloned methods, which may indicate that developers are careful to copy-paste only small(er) pieces of code. Additionally, there are only three metrics *NEXP*, *HEFF*, and *XMET* where the cloned methods are significantly different than the non-cloned methods. Also, our observations resonate with other studies in finding that size is a confounding variable in such studies. Similar to what is described in [100], normalizing the metrics for size does not always make the metrics completely independent of size.

We performed a mixed method analysis to understand why the cloned methods differ from the non-cloned methods with respect to the size metric. We found one possible factor that may lead to the size differences– presence of many small auto-generated methods that are deemed clones because of their similar structure. Additionally, we gained some more insights about cloned methods. Specifically, among the clones that do not seem to be automatically generated, we found many instances of multi-purpose methods, i.e. methods that have very similar structure and vocabulary but do the opposite (expand vs. collapse, etc.). We also looked into if cloned and non-cloned methods perform fundamentally different functions, and found that certain functionality tends to be more pervasive in cloned methods than non-cloned methods. Additionally, we manually evaluated the comments and found no statistically significant difference between the cloned and non-cloned methods with respect to the usability of comments. Also, the cloned and non-cloned methods showed no statistically significant difference with respect to their comprehensibility to human observers.

Dataset accessibility and Reproducibility: We have made available all the necessary artifacts including raw data, detailed steps and scripts to process data, and analysis procedure to reproduce the statistical results to verify the claims. All artifacts are made available at: `http://mondego.ics.uci.edu/projects/clone-metrics`.

3.2 Study 2. How Much of GitHub is Duplicated?

This study explores the extent of cloning in Github, a popular software repository hosting online service. The study was conducted for projects written in four popular programming languages, namely, Java, Javascript, C++, and Python.

In the following sections, I will briefly describe the key aspects of this study including, the contributions, dataset, and summary of quantitative and qualitative analyses of our results along with the analysis pipeline. I will also describe the challenges we faced owing to the scale of our study, and how we addressed them.

Table 3.9: File-hash duplication in subsets.

	10K Stars	10K Commits
Java	9%	6%
C/C++	41%	51%
Python	28%	44%
JavaScript	44%	66%

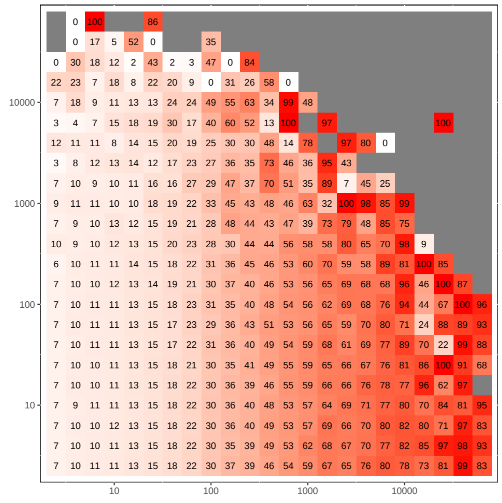
We expected to answer the following questions: How much code cloning is there, how does cloning affect datasets of software written in different languages, and through which processes does duplication come about?

We conducted this study at three levels of similarity. A *file hash* gives a measure of file that are copied across projects without changes. A *token hash* captures minor changes in spaces, comments and ordering. Lastly, SourcererCC captures files with 80% token-similarity. This gives an idea of how many files have been edited after cloning.

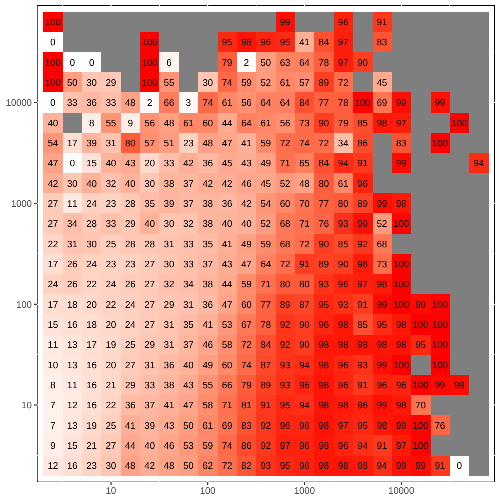
3.2.1 Contributions

This study provides a tool to assist selecting projects from GitHub. DéjàVu is a publicly available index of file-level code duplication. The novelty of this work lies partly in its scale; it is an index of duplication for the entire GitHub repository for four popular languages, Java, C++, Python and JavaScript. Figure 3.26 illustrates the proportion of duplicated files for different project sizes and numbers of commits (Section 3.2.4 explains how these heatmaps were generated). The heatmaps show that as project size increases the proportion of duplicated files also increases. Projects with more commits tend to have fewer project-level clones. Finally JavaScript projects have the most project-level clones, while Java projects have the fewest.

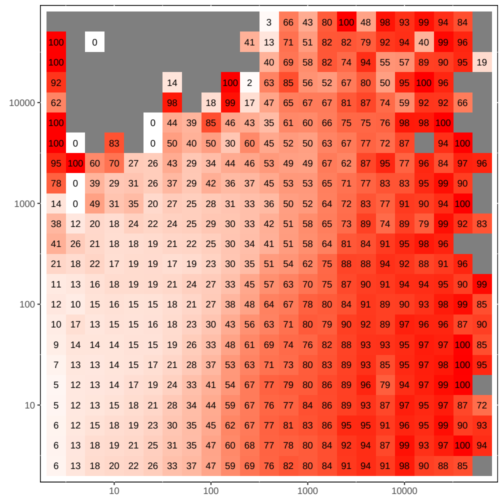
The clone map from which the heatmaps were produced is the main contribution of this study. It can be used to understand the similarity relations in samples of projects or to curate samples to reduce duplicates. DéjàVu can be used to curate datasets, i.e. remove projects with too many clones. Besides applicability to research, our results can be used by anyone who needs to host large amounts of source code to avoid storing duplicate files. Our clone map can also be used to improve tooling, e.g. being queried when new files are added to projects to filter duplicates.



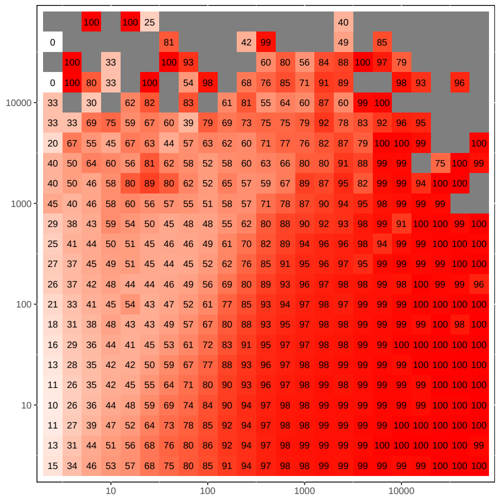
Java



Python



C++



JavaScript

Figure 3.26: Map of code duplication. The y-axis is the number of commits per project, the x-axis is the number of files in a project. The value of each tile is the percentage of duplicated files for all projects in the tile. Darker means more clones.

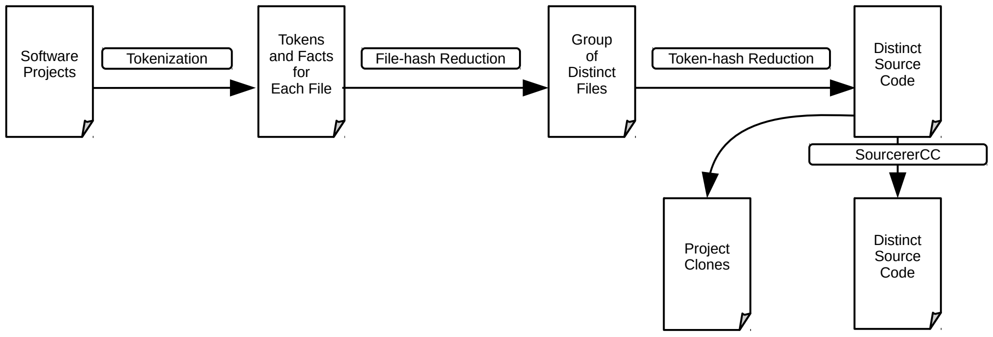


Figure 3.27: Analysis pipeline.

3.2.2 Analysis Pipeline

Our analysis pipeline is outlined in Figure 3.27. The pipeline starts with local copies of the projects that constitute our corpus. From here, code files are scanned for fact extraction and tokenization. Two of the facts are the hashes of the files and the hashes of the tokens of the files. File hashes identify exact duplicates; token hashes allow catch clones up with minor differences. While permutations of same tokens may have the same hash, they are unlikely. Clones are dominated by exact copies, and we did not observe any such collision in randomly sampled pairs. Files with distinct token hashes are used as input to the near-miss clone detection tool, SourcererCC. While our JavaScript pipeline was developed independently, data formats, database schema and analysis scripts are identical.

3.2.2.1 Tokenization

Tokenization transforms a file into a “bag of words,” where occurrences of each word are recorded. Consider, for instance, the Java program:

```java
package foo;
public class Foo {  // Example Class
   private int x;
   public Foo(int x) { this.x = x; }
   private void print() { System.out.println("Number: " + x) }
   public static void main() {  new FooNumber(4).print(); }
}
```

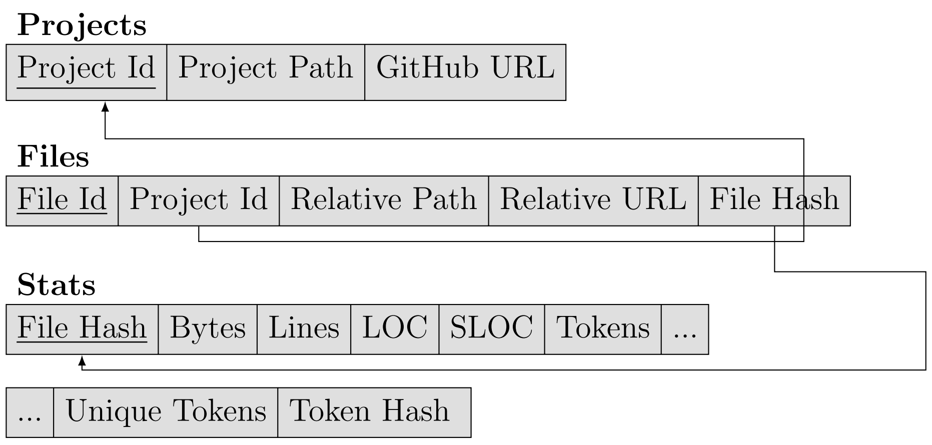
Tokenization removes comments, white space, and terminals. Tokens are grouped by frequency, generating:

```
Java Foo:[(package,1),(foo,1),(public,3),(class,1),(Foo,2),(private,2),(int,2),(x,5),
(this,1),(void,2),(print,2),(System,1),(out,1),(println,1),(Number,1),(static,1),
(main,1),(new,1),(FooNumber,1),(4,1)]
```

The tokens `package` and `foo` appear once, `public` appears three times, etc. The order is not relevant. During tokenization we also extract additional information: (1) *file hash* – the MD5 hash of the entire string that composes the input file; (2) *token hash* – the MD5 hash of the string that constitutes the tokenized output; (3) *size* in bytes; (4) *number of lines*; (5) *number of lines of code* without blanks; (6) *number of lines of source* without comments; (7) *number of tokens*; and (8) *number of unique tokens*. The tokenized input is used both to build a relational database and as input to SourcererCC. The use of MD5 (or any hashing algorithm) runs the risk of collisions, given the size of our data they are unlikely to skew the results.

3.2.2.2 Database

The data extracted by the tokenizer is imported into a MySQL database. The table `Projects` contains a list of projects, with a unique identifier, a path in our local corpus and the project's URL. `Files` contains a unique id for a file, the id of the project the file came from, the relative paths and URLs of the file and the file hash. The statistics for each file are stored in the table `Stats`, which contains the information extracted by the tokenizer. The tokens themselves



are not imported. The `Stats` table has the file hash as unique key. With this, we get an immediate reduction from files to hash-distinct files. Two files with distinct file hashes may produce the exact same tokens, and, therefore the same token hash. This could happen when the code of one file is a permutation of another. The converse does not hold: files with distinct token hashes must have come from files with distinct file hashes. For source code analysis, file hashes are not necessarily the best indicators of code duplication; token hashes are more robust to small perturbations. We use primarily token hashes in our analysis.

3.2.2.3 Project-Level Analysis

Besides file-level analysis, we also look for projects with significant overlap with other projects. This is done with a script that queries the database making an intersection of the project files' distinct token hashes. This script produces pairs of projects that have significant overlap in at least one direction. The results are of the form: *A cloned in B at x%, B cloned in A at y%*, where $x\%$ of project A's files (in tokenized form) are found also in project B, and $y\%$ of project B's files (in tokenized form) are found in project A. Calculating project-level information is done in two steps. First, collect all the files from a project A, say, for example there are 4 files in A: Then find the token-hash duplicates for each of these files in other projects. It might be something like:

```
project A
File_1 − B, B, C
File_2 − B
File_3 −
File_4 − B, D, F
```

There are 3 files from `A` with duplicates in `B`, making `A` a clone of `B` at 75%. Conversely, there are 4 files in `B` with duplicates in `A`; assuming `B` has a total of 20 files, then `B` is cloned in `A` at 20%. A file can be in other project multiple times (e.g. in different directories) as is File 1.

3.2.2.4 SourcererCC

The concept of inexact code similarity has been studied in the code cloning literature. Blocks of code that are similar are called near-miss clones, or near-duplication [126]. SourcererCC estimates the amount of near-duplication in GitHub with a "bag of words" model for source code rather than more sophisticated structure-aware clone detection methods. It has been shown to have good precision and recall, comparable to more sophisticated tools [133]. A part of the reason why we decided to use SourcererCC as our tool of choice was because SourcererCC has been demonstrated to be the most scalable near-miss Type III clone detector publicly available at that time. It has been demonstrated that SourcererCC can successfully scale to 250 million lines of code across 25,000 Java projects. Its input consists of non-empty files with distinct token hashes. SourcererCC finds clone pairs between these files at a given level of similarity. We have selected 80% similarity as this has given good empirical results. Ideally one could imagine varying the level of similarity and reporting a range of results. But this would be computationally expensive and, given the relatively low numbers of near-miss clones, would not affect our results.

Table 3.10: GitHub Corpus.

		Java	C++	Python	JavaScript
Counts	# projects (total)	3,506,219	1,130,879	2,340,845	4,479,173
	# projects (non-fork)	1,859,001	554,008	1,096,246	2,011,875
	# projects (downloaded)	1,481,468	369,440	909,290	1,778,679
	# projects (analyzed)	1,481,468	364,155	893,197	1,755,618
	# files (analyzed)	72,880,615	61,647,575	31,602,780	261,676,091
Medians	Files/project	9 ($\sigma = 600$)	11 ($\sigma = 1304$)	4 ($\sigma = 501$)	6 ($\sigma = 1335$)
	SLOC/file	41 ($\sigma = 552$)	55 ($\sigma = 2019$)	46 ($\sigma = 2196$)	28 ($\sigma = 2736$)
	Stars/project	0 ($\sigma = 71$)	0 ($\sigma = 119$)	0 ($\sigma = 99$)	0 ($\sigma = 324$)
	Commits/project	4 ($\sigma = 336$)	6 ($\sigma = 1493$)	6 ($\sigma = 542$)	6 ($\sigma = 275$)

3.2.3 Corpus

The GitHub projects were downloaded using the GHTorrent database and network [50] which contains meta-data such as number of stars, commits, committers, whether projects are forks, main programming language, date of creation, etc., as well as download links. While convenient, GHTorrent has errors: 1.6% of the projects were replicated entries with the same URL; only the youngest of these was kept for the analysis.

Table 3.10 gives the size of the different language corpora. We skipped forked projects as forks contain a large amount of code from the original projects, retaining those would skew our findings. Downloading the projects was the most time-consuming step. The order of downloads followed the GHTorrent projects table, which seems to be roughly chronological. Some of the URLs failed to produce valid content. This happened in two cases: when the projects had been deleted, or marked private, and when development for the project happens in branches other than master. Thus, the number of downloaded projects was smaller than the number of URLs in GHTorrent. For each language, the files analyzed were files whose extensions represent source code in the target languages. For Java: `.java`; for Python: `.py`; for JavaScript: `.js`, for C/C++: `.cpp .hpp .HPP .c .h .C .cc .CPP .c++` and `.cp`. Some projects did not have any source code with the expected extension, they were excluded.

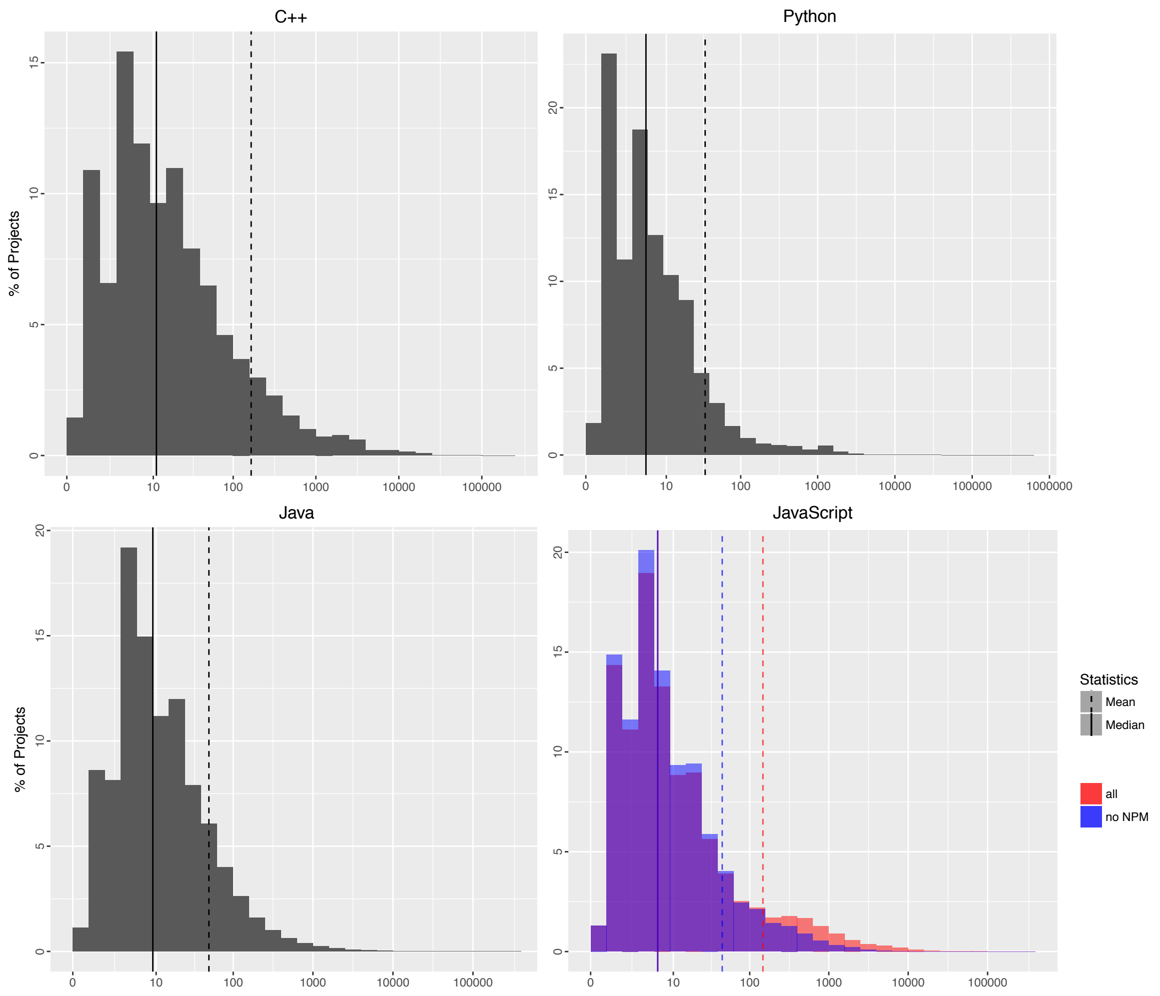


Figure 3.28: Files per project.

The medians in Table 3.10 give additional properties of the corpus, namely the number of files per (non-empty) project, the number of Source Lines of Code (SLOC) per file, the number of stars and the number of commits of the projects. In terms of files per project, Python and JavaScript projects tend to be smaller than Java and C++ projects. C++ files are considerably larger than any others, and JavaScript files are considerably smaller. None of these numbers is surprising. They all confirm the general impression that a large number of projects hosted in GitHub are small, not very active, and not very popular. Figures 3.28 and 3.29 illustrate the basic size-related properties of the projects we analyzed, namely the distribution of files per project and the distribution of Source Lines of Code (SLOC) per file. For JavaScript we give data with and without NPM (it is a cause of a large number

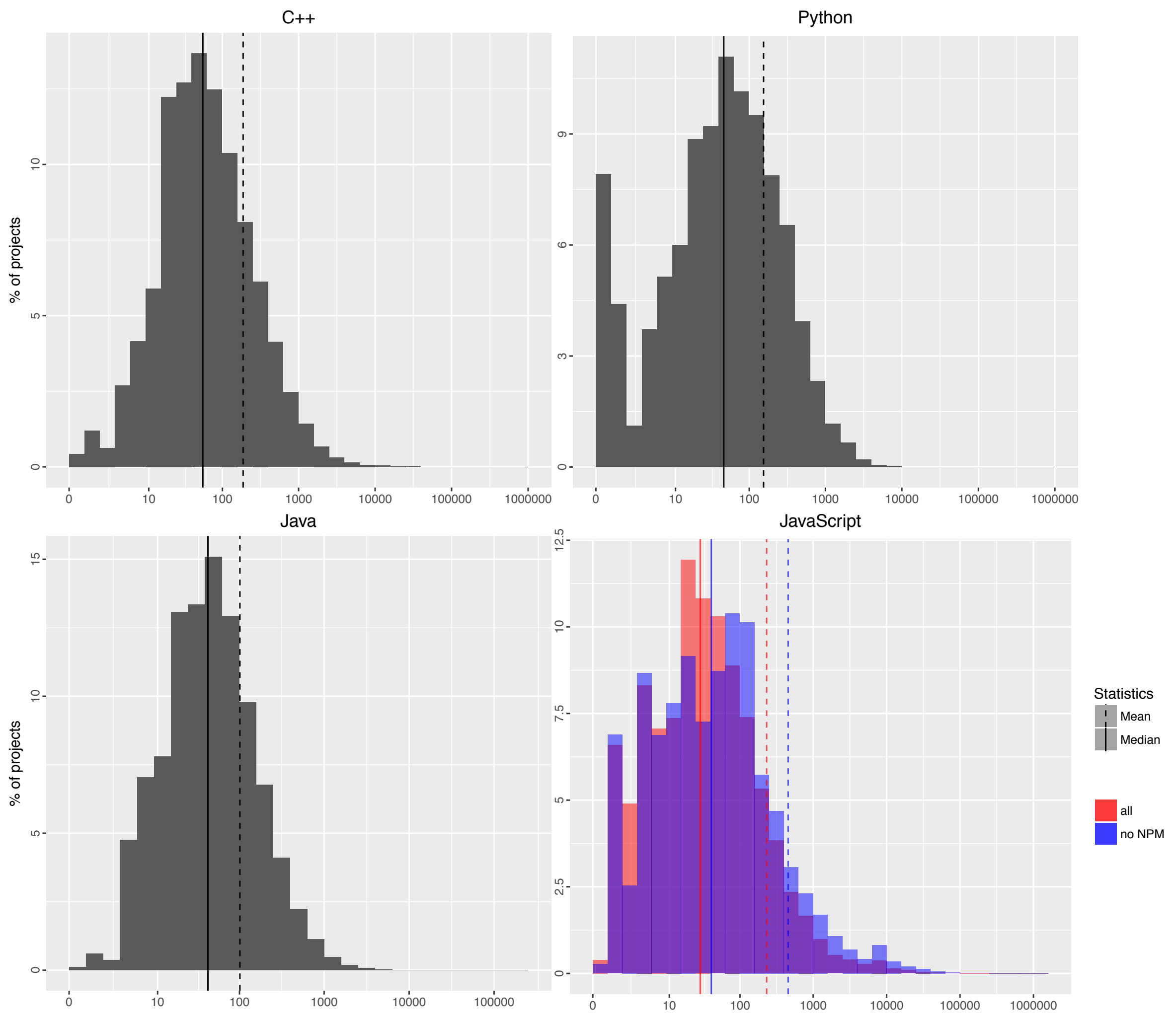


Figure 3.29: SLOC per file.

of clones). Without NPM means that we ignored files downloaded by the Node Package Manager.

3.2.4 Summary of Results

We present analyses of the data at two levels of detail: file and project level.

3.2.4.1 File-Level Analysis

Table 3.11 shows a summary of the findings for files. “SCC dup files” is the number of files, out of the distinct token-hash files, that SourcererCC has identified as clones; similarly, “SCC unique files” is the number of files for which no clones were detected. Figure 3.30 (top row) charts the numbers in Table 3.11. The duplicated files (dark grey) are the files that are duplicate of at least one of the distinct token-hash files (light grey); further, the distinct token-hash files are split between those for which SourcererCC found at least one similar file (cloned files, grey) and those for which SourcererCC did not find any similar file (unique files, in white).

The amount of duplication varies with the language: the JavaScript ecosystem contains the largest amount of duplication, with 94% of files being file-hash clones of the other 6%; the Java ecosystem contains the smallest amount, but even for Java, 40% of the files are duplicates; the C++ and Python ecosystems have 73% and 71% copies, respectively. As for near-duplicates, Java contains the largest percentage: 46% of the files are near-duplicate clones. The ratio of near-miss clones is 43% for Java, 39% for JavaScript, and 32% for Python.

Table 3.11: File-Level Duplication.

	Java	C++	Python	JavaScript
Total files	72,880,615	61,647,575	31,602,780	261,676,091
File hashes	43,713,084 (60%)	16,384,801 (27%)	9,157,622 (29%)	15,611,029 (6%)
Token hashes	40,786,858 (56%)	14,425,319 (23%)	8,620,326 (27%)	13,587,850 (5%)
SCC dup files	18,701,593 (26%)	6,200,301 (10%)	2,732,747 (9%)	5,245,470 (2%)
SCC unique files	22,085,265 (30%)	8,225,018 (13%)	5,887,579 (19%)	8,342,380 (3%)

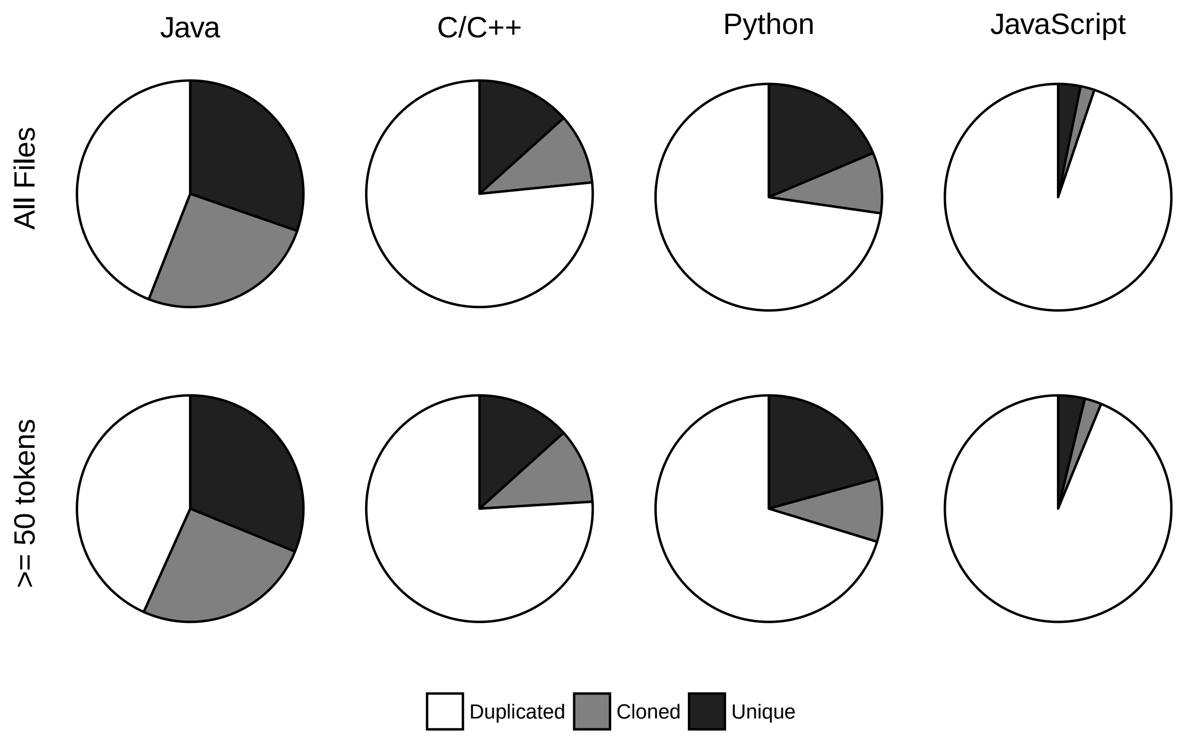


Figure 3.30: File-level duplication for entire dataset and excluding small files.

3.2.4.2 File-Level Analysis Excluding Small Files

We redid our analysis after excluding smaller files to remove files that are trivial (empty) or do not contain enough code to represent any specific functionality. Specifically, we excluded all files with less than 50 tokens.[2] Table 3.12 and Figure 3.30 (bottom row) show the results.

Although the absolute number of files and hashes change significantly, the changes in ratios

[2]This threshold is arbitrary. It is based on our observations of small files; other values can be used.

Table 3.12: File-level duplication excluding small files.

	Java	C++	Python	JavaScript
# of files	57,240,552	49,507,006	23,382,050	162,136,892
% of corpus	79%	80%	74%	62%
File hashes	34,617,736 (60%)	13,401,948 (27%)	7,267,097 (31%)	11,444,667 (7%)
Token hashes	32,473,052 (58%)	11,893,435 (24%)	6,949,894 (30%)	10,074,582 (6%)
SCC dup files	14,626,434 (26%)	5,297,028 (10%)	2,105,769 (9%)	3,896,989 (2%)
SCC unique files	17,848,618 (31%)	6,596,407 (13%)	4,844,125 (21%)	6,177,593 (4%)

of the hashes and SCC results are small. When they are noticeable, they show that there is slightly less duplication in this dataset than in the entire dataset. Comparing Table 3.12 with Table 3.11 shows that small files account for a slightly higher presence of duplication, but not that much higher than the rest of the corpus.

3.2.4.3 Inter-Project Analysis

Next, we looked into inter-project cloning, i.e., how many projects are exact and near-duplicates of other projects, even though they are not technically *forks*? The results are shown in Table 3.13 and Figure 3.31.

Table 3.13: Inter-project cloning.

	Java	C++	Python	JavaScript
# projects (analyzed)	1,481,468	364,155	893,197	1,755,618
# clones ≥ 50%	205,663 (14%)	94,482 (25%)	159,224 (18%)	854,300 (48%)
# clones ≥ 80%	135,168 (9%)	58,906 (16%)	94,634 (11%)	546,207 (31%)
# clones 100%	87,220 (6%)	24,851 (7%)	51,589 (6%)	273,970 (15%)
# exact dups	73,869 (5%)	19,809 (5%)	43,501 (5%)	198,556 (11%)
# exact dups (≥ 10 files)	37,722 (3%)	10,286 (3%)	7,331 (1%)	78,972 (4%)

Table 3.13 shows the number of projects whose files exist in other projects at some overlap threshold – 50%, 80% and 100%, respectively. A normalization of these numbers over the total number of projects for each language is shown in Figure 3.31. JavaScript comes on top with respect to the amount of project-level duplication, with 48% of projects having 50% or more files duplicated in some other project, and an equally impressive 15% of projects being 100% duplicated.

The last two rows of Table 3.13 show the number of projects that are token-hash clones of some other project (apart from differences in white space, comments, and terminal symbols). While all other languages show 5% project level duplication (exact), the duplication in javascript is 11%, almost double of what observed in other languages.

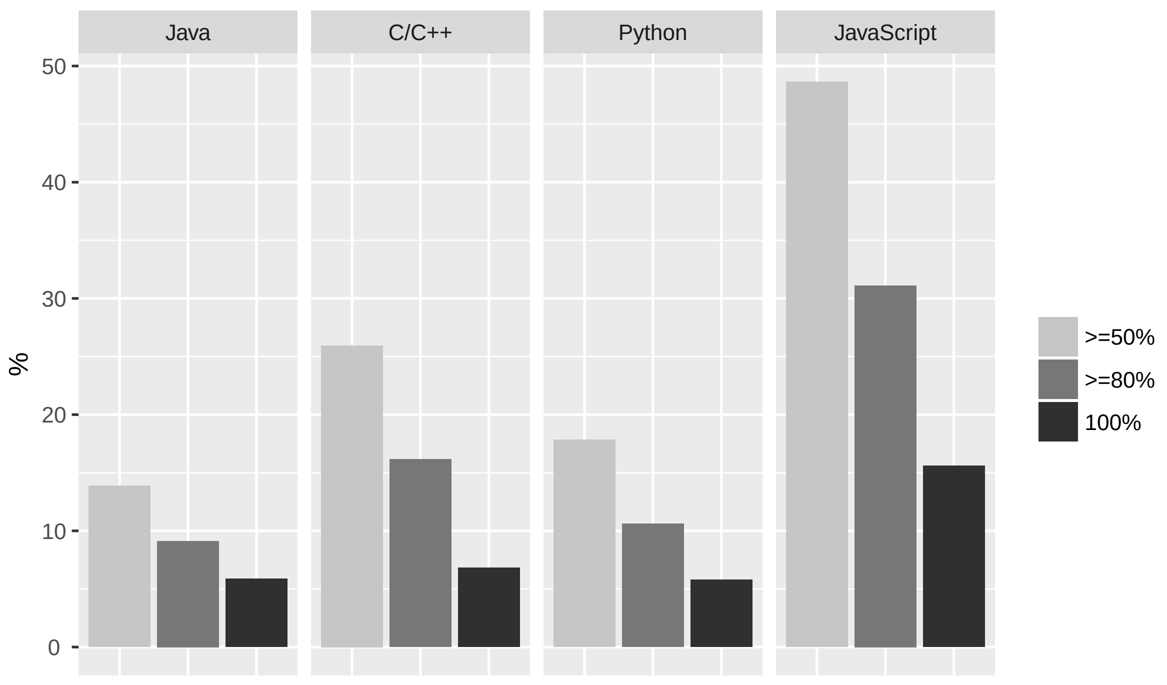


Figure 3.31: Percentage of project clones at various levels of overlap.

3.2.5 SourcererCC Performance Improvements

As mentioned earlier, in this study, we explored the extent of cloning in the software project repositories hosted on GitHub. The study includes projects written in four popular programming languages, namely, Java, Python, C/C++, and JavaScript. In total, we analyzed 427,807,061 files across 4,494,438 projects. The size of our dataset posed challenges related to both scalability and execution time. To be able to use SourcererCC at this scale in a reasonable time, we identified and addressed performance bottlenecks in the design of SourcererCC. In this section, I describe these performance bottlenecks and how we addressed them.

Though SourcererCC was the fastest near-miss scalable clone detector available at the time of this study, it was not fast enough to complete clone detection on our dataset in a reasonable time. We identified large number of expensive I/O operations to be the major reason behind the slowness of SourcererCC at this scale. These I/O bottlenecks are described below.

- SourcererCC creates the inverted index of code blocks in order to speed up the process of clone detection. With the increase in the size of dataset, the size of inverted index also increases, making it difficult to load this index in memory. Having an on-disk index slows down SourcererCC as every query to the index leads to a disk I/O operation.

- One good approach to speed up the execution time is to create multiple partitions of a large query file such that, each of these partitions contains a subset of queries from the given large query file. One can then run multiple instances of SourcererCC to detect clones for each of these smaller partitioned query files. This approach improves the execution time by running several queries in parallel. Each of the instances of SourcererCC, however, requires the entire inverted index created using the complete dataset. And therefore, in this approach, the issue of expensive I/O remains unresolved.

- SourcererCC creates a global map of tokens and their frequencies in the given dataset. For large datasets, this global map increases in size and therefore cannot be stored in memory. To compare if a code block pair is a clone, SourcererCC makes several look-ups to this map which further increases disk I/O.

To address the above mentioned issues, we introduced the following design changes in SourcererCC.

- Size-based Input Partitioning. We partition the dataset into multiple partitions based on the size of the code blocks. We divide the dataset into multiple partitions, such that each partition contains only those code blocks which are within certain lower and the upper size limits. This size-based input partitioning is explained in Section 4.2.1.2. One key benefit of this partitioning strategy is that it allows us to identify all of the queries that need to be run on a specific partition. Moreover, any given query needs to be run only on one partition, making the partitions mutually exclusive.

We modified SourcererCC to make these mutually exclusive partitions. Multiple instances of SourcererCC can now be run on these mutually exclusive partitions, such that each instance of SourcererCC is responsible for creating and using the index of the code blocks present in its partition. The modification resulted into the creation of smaller indexes making them easier to load in the memory and save large amount of expensive I/O operations.

- Index Partitioning. Another important design modification is to use a second-level size-based partition of index within each top-level partition. We modified SourcererCC to have an upper bound on the size of the in-memory index. With this modification, SourcererCC creates an in-memory index of the blocks in its partition till it reaches the upper bound. It then queries this in-memory index for all of the queries in its partition to detect clones. Next, it discards the earlier index and creates another in-memory index of the remaining blocks in its partition and queries it again with all of the queries in the partition. This process goes on till all of the blocks in the partition are indexed. This strategy leads to fast in-memory lookups, thereby increasing the overall speed of clone-detection.

- Caching. SourcererCC creates and uses a global token-frequency map to compare the tokens of a candidate clone pair. To speed up the look up to this map, we created an in-memory LRU cache, a common strategy to avoid expensive disk or network I/O lookups.

The above design changes allowed us to run multiple instances of SourcererCC, each working on a smaller subset of the dataset. Moreover, the design changes reduced the amount of expensive disk I/O operations significantly. We compared the runtime performance of modified SourcereCC (InMemCC) with the published version of SourcererCC on IJaDataset [4], a large inter-project Java repository containing 25,000 open-source projects (3 million source files, 250MLOC) mined from SourceForge and Google Code. Only one instance was created for each SourcererCC and InMemCC to carry out this experiment.

The modified SourcererCC took 4 hours and 40 minutes to complete the execution whereas the published version of SourcererCC took 96 hours to complete clone detection. The design changes lead one (close to two) order of magnitude improvement in the execution time of SourcererCC. The changes enabled us to use the core engine of SourcererCC to detect clones much faster on very large datasets. The changes related to caching were led by Prof. Cristina Lopes, whereas I led the changes related to input partitioning and index partitioning.

3.2.6 Conclusions

The source control system upon which GitHub is built, Git, encourages forking projects and independent development of those forks. GitHub provides an easy interface for forking a project, and then for merging code changes back to the original projects. This is a popular feature: the metadata available from GHTorrent shows an average of 1 fork per project. However, there is a lot more duplication of code that happens in GitHub that does not go through the fork mechanism, and, instead, goes in via copy and paste of files and even entire libraries.

We presented an investigation of code cloning in GitHub for four of the most popular object-oriented languages: Java, C++, Python and JavaScript. The amount of file-level duplication is staggering in the four language ecosystems, with the extreme case of JavaScript, where only 6% of the files are original, and the rest are copies of those. The Java ecosystem has the least amount of duplication. These results stand even when ignoring very small files. When delving deeper into the data we observed the presence of files from popular libraries that were copy-included in a large number projects.

We also presented some of the scalability challenges we faced while working with SourcererCC and how we addressed them.

Chapter 4

Oreo: Scalable and Accurate Clone Detection in the Twilight Zone

Part of the material in this chapter and the following chapter is included with the permission of ACM and based on our work in:

- Vaibhav Saini, Farima Farmahinifarahani, Yadong Lu, Pierre Baldi, and Cristina V. Lopes. 2018. Oreo: Detection of Clones in the Twilight Zone. In Proceedings of the 26th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '18), November 4–9, 2018, Lake Buena Vista, FL, USA. ACM, New York, NY, USA, 12 pages. `https://doi.org/10.1145/3236024.3236026`

  This work received **ESEC/FSE 2018 Distinguished Paper Award.** Also, it received three badges by the artifact evaluation committee at ESEC/FSE '18: i) Available, ii) Reusable, and iii) Functional.

4.1 Introduction

This chapter introduces Oreo, a scalable method-level clone detector that is capable of detecting not just Type-1 through strong Type-3 clones, but also clones in the Twilight Zone. In our experiments, the recall values for Oreo are similar to other state of the art tools in detecting Type-1 to strong Type-3 clones. However, Oreo performs much better on clones where the syntactic similarity reduces below 70% – the area of clone detection where the vast majority of clone detectors do not operate. The number of these harder-to-detect clones detected by Oreo is one to two orders of magnitude higher than the other tools. Moreover, Oreo is scalable to very large datasets.

The key insights behind the development of Oreo are twofold: (1) functionally similar pieces of code tend to do similar *actions*, as embodied in the functions they call and the state they access; and (2) not all pieces of code that do similar actions are functionally similar; however, it is possible to *learn*, by examples, a combination of metric weights that can predict whether two pieces of code that do the same actions are clones of each other. For semantic similarity, we use a novel *Action Filter* to filter out a large number of method pairs that don't seem to be doing the same actions, focusing only on the candidates that do. For those potential clones, we pass them through a supervised machine learning model that predicts whether they are clones or not. A deep learning model is trained based on a set of metrics derived from source code. We demonstrate that Oreo is accurate and scalable.

The results presented in this paper were obtained by training the metrics similarity model using SourcererCC [137], a state of the art clone detector that has been shown to have fairly good precision and recall up to Type-3 clones (but not Type-4). However, our approach is not tied to SourcererCC; any accurate clone detector can be used to train the model. Specifically, many clone detection techniques like graph-based or AST-based, which are accurate but hard to scale, can be used in the training phase.

The contributions of this paper are as follows:

- **Detection of clones in the Twilight Zone**. Compared to reported results of other clone detectors in the literature, Oreo's performance on harder-to-detect clones is the best so far.

- **Analysis of clones in the Twilight Zone**. Besides quantitative results, we present analysis of examples of harder-to-detect clones – a difficult task, even for humans, of deciding whether they are clones, and the reasons why Oreo succeeds where other clone detectors fail.

- **Process-pipeline to learn from slow but accurate clone detector tools and scale to large datasets**. The clone detection techniques which are accurate but hard to scale can be used to train a model and predict clones in a scalable manner using the concepts introduced in this paper.

- **Deep Neural network with Siamese architecture**. We propose Siamese architecture [9] to detect clone pairs. An important characteristic of this architecture is that it can handle the symmetry [109] of its input vector (presenting the pair (a, b) to the model will be the same as presenting the pair (b, a), a desirable property in clone detection).

The remainder of this chapter is organized as follows. Section 4.2 presents three concepts that are parts of our proposed approach and are critical to its performance; Section 4.3 explains the deep neural network model used in our approach and how it was selected and configured; Section 4.2.3 describes the clone detection process using the concepts introduced in Sections 4.2 and 4.3.

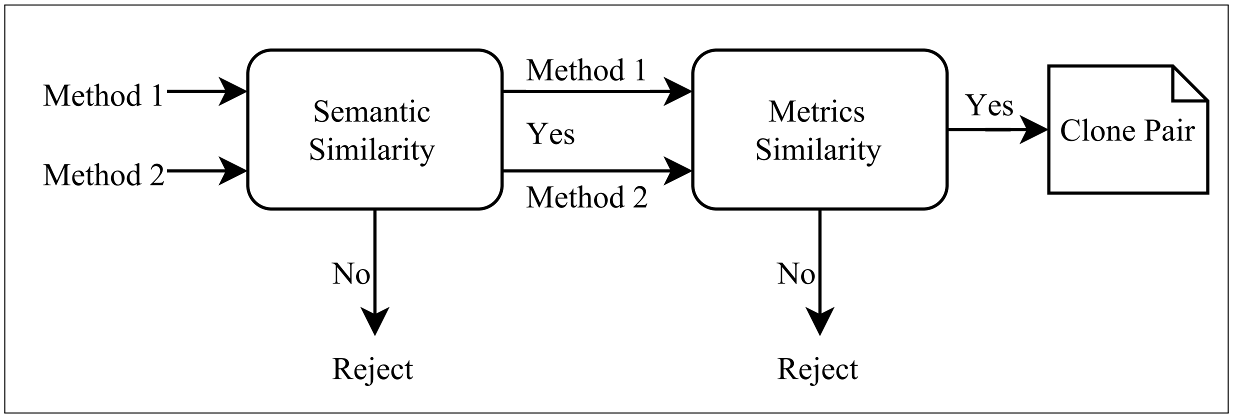


Figure 4.1: Overview of Oreo.

4.2 The Oreo Clone Detector

The goals driving the design of Oreo are twofold: (1) we want to be able to detect clones in the Twilight Zone without hurting precision, and (2) we want to be able to process very large datasets consisting of hundreds of millions of methods. In order to accomplish the first goal, we introduce the concept of semantic signature, which is based on actions performed by that method, followed by an analysis of the methods' software metrics. In order to accomplish the second goal, we first use a simple size-based heuristic that eliminates a large number of unlikely clone pairs. Additionally, the use of semantic signatures also allows us to eliminate unlikely clone pairs early on, leaving the metrics analysis to only the most likely clone pairs. Figure 4.1 gives an overview of Oreo.

4.2.1 Scalability

The performance improvement we did for SourcererCC, as explained in Section 3.2.5 forms the basis of making Oreo a scalable clone detector. To design Oreo we use the size-based input partitioning which allowed us to create in-memory indexes for faster lookups. In this section I first explain the preprocessing step which allowed us to carry out faster processing during clone detection. Then, I explain how Oreo uses size similarity sharding and in-memory indexes to achieve high scalability.

4.2.1.1 Preprocessing

One key strategy to scaling super-linear analysis of large datasets is to preprocess the data as much as possible. Preprocessing consists of a one-time, linear scan of the data with the goal of extracting features from it that allow us to better organize and optimize the actual data processing. In Oreo, during preprocessing, we extract several pieces of information about the methods, namely: (1) their semantic signature (Section 4.2.1.3), and (2) assorted software metrics.

Table 4.1 shows the 24 method level metrics extracted from the source files. A subset of these metrics is derived from the Software Quality Observatory for Open Source Software (SQO-OSS) [138]. The decision of, which SQO-OSS metric to include is based on one simple condition: a metric's correlation with the other metrics should not be higher than a certain threshold. This was done because two highly correlated metrics will convey very similar information, making the presence of one of them redundant. From a pair of two correlated metrics, we retain the metric that is faster to calculate.

Additionally to SQO-OSS, we extract a few more metrics that carry important information. During our initial exploration of clones in the Twilight Zone, we noticed many clone pairs where both methods are using the same type of literals even though the literals themselves are different. For example, there are many cases where both the methods are using either *Boolean* literals, or *String* literals. Capturing the *types* of these literals is important as they contain information that can be used to differentiate methods that operate on different types – a signal that they may be implementing different functionality. As a result, we add a set of metrics (marked with $*$ in the Table 4.1) that captures the information on how many times each type of literal is used in a method. A description of many of these software metrics is

Table 4.1: Method-Level Software Metrics

Name	Description	Name	Description
XMET	# external methods called	HEFF	Halstead effort to implement
VREF	# variables referenced	HDIF	Halstead difficulty to implement
VDEC	# variables declared	EXCT	# exceptions thrown
NOS	# statements	EXCR	# exceptions referenced
NOPR	# operators	CREF	# classes referenced
NOA	# arguments	COMP	McCabes cyclomatic complexity
NEXP	# expressions	CAST	# class casts
NAND	# operands	NBLTRL∗	# Boolean literals
MDN	maximum depth of nesting	NCLTRL∗	# Character literals
LOOP	# loops (for,while)	NSLTRL∗	# String literals
LMET	# local methods called	NNLTRL∗	# Numerical literals
HVOC	Halstead vocabulary	NNULLTRL∗	# Null literals

given in Chapter 3.

4.2.1.2 Size Similarity Sharding

When doing clone detection on real code, the vast majority of method pairs are *not* clones of each other. However, the clone detector needs to process all possible pairs of methods in order to find out which ones are clones. This can be extremely costly, and even prohibitive on very large datasets, when the technique used for detecting clones is CPU-intensive. A general strategy for speeding up clone detection is to aggressively eliminate unlikely clone pairs upfront based on very simple heuristics. We call the pairs which survive this aggressive elimination as *candidate pairs*. Any method for which we are detecting clones is a *query* and the methods which form a candidate pair with a *query* are called *candidate clones* of that *query*.

The first, and simplest, heuristic used by Oreo is size. The intuition is that two methods with considerably different sizes are very unlikely to implement the same, or even similar, functionality. This heuristic can lead to some false negatives, specifically in the case of Type-4 clones. However, in all our experiments, we observed little to no impact on the recall of other clone types.

As a metric of method size we use the number of tokens in the method, where tokens are language keywords, literals (string literals are split on whitespace), types, and identifiers. This is the same definition used in other clone detection work (e.g. [137]). Given a similarity threshold T between 0 and 1, and a method M_1 with x tokens, if a method M_2 is a clone of M_1, then its number of tokens, y, should satisfy the inequation $x \times T \leq y \leq \frac{x}{T}$.

In Oreo, this size similarity filter is implemented in the preprocessing phase, by partitioning the dataset into shards based on the size of the methods. We divide the dataset into multiple partitions, such that each partition contains only methods within certain lower and the upper size limits. In a partition, the methods with size between the upper and the lower size limits of the partition get to be the query methods. The partition's lower and upper limits for candidate methods are calculated using the inequation given above, where x is substituted with the partition's lower and upper limits. The partitions are made such that any given candidate method will at most belong to two partitions. The remaining computations for clone detection are performed exclusively within each of the shards.

Besides acting as a static filter for eliminating unlikely clones, size-based sharding is also the basis for the creation of indexes that speed up clone detection in subsequent filters.

Another important design detail is that Oreo uses a second-level size-based sharding within each top-level shard, for the purpose of loading batches of candidate pairs into memory. During clone detection, we load each second-level shard into the memory one by one and query it with all query methods in the shard's parent partition. This leads to fast in-memory lookups, thereby increasing the overall speed of clone-detection. The idea of input partitioning is not new, and has been used in information retrieval systems many times [25, 90, 96]. Researchers in other fields have explored partitioning based on size and also horizontal partitioning to solve the scalability and speed issues [95]. Here, we apply those lessons to our clone detector.

4.2.1.3 Semantic Similarity: The Action Filter

Clones in the Twilight Zone have low lexical and syntactic similarity, but still perform similar functions. In order to detect clones in this spectrum, some sort of semantic comparison is necessary. We capture the semantics of methods using a semantic signature consisting of what we call *Action tokens*. The *Action tokens* of a method are the tokens corresponding to methods called and fields accessed by that method. Additionally, we capture array accesses (e.g. filename[i] and filename[i+1]) as *ArrayAccess* and *ArrayAccessBinary* actions, respectively. This is to capture this important semantic information that Java encodes as syntax.

Semantic signatures are extracted during preprocessing. As an example of *Action tokens* extraction, consider the code in Listing 4.1, which converts its input argument to an encrypted format. The resulting *Action tokens* are: *getBytes()*, *getInstance()*, *update()*, *digest()*, *length*, *append()*, *toString()*, *traslate()*, *ArrayAccess*, and *toString()*.[1] More than the identifiers chosen by the developer, or the types used, these *Action tokens* form a better semantic signature of the method. The intuition is that if two methods perform the same function, they likely call the same library methods and refer the same object attributes, even if the methods are lexically and syntactically different. Modern libraries provide basic semantic abstractions that developers are likely to use; Oreo assumes the existence and use of these abstractions. Hence, we utilize these tokens to compare semantic similarity between methods. This is done in the first dynamic filter of Oreo, the *Action filter*. We use overlap-similarity, calculated as $Sim(A_1, A_2) = |A_1 \cap A_2|$, to measure the similarity between the *Action tokens* of two methods. Here, A_1 and A_2 are sets of Action Tokens in methods M_1 and M_2 respectively. Each element in these sets is defined as $< t, freq >$, where t is the Action Token and $freq$

[1]The *ArrayAccess action token* stands for *hashedPasswd[i]*.

is the number of times this token appears in the method.

Listing 4.1: Action Filter Example

```java

public static String getEncryptedPassword(String password) throws InfoException {
    StringBuffer buffer = new StringBuffer();
    try {
        byte[] encrypt = password.getBytes("UTF-8");
        MessageDigest md = MessageDigest.getInstance("SHA");
        md.update(encrypt);
        byte[] hashedPasswd = md.digest();
        for (int i = 0; i < hashedPasswd.length; i++) {
            buffer.append(Byte.toString(hashedPasswd[i]));
        }
    } catch (Exception e) {
    throw new InfoException(LanguageTraslator.traslate("474"), e);
}
return buffer.toString();
}
```

In order to speed up comparisons, we create an inverted index of all the methods in a given shard using *Action tokens*. To detect clones for any method, say M, in the shard, we query this inverted index for the *Action tokens* of M. Any method, say N, returned by this query becomes a candidate clone of M provided the overlap-similarity between M and N is greater than a preset threshold. We call M the query method, N a candidate of M, and the pair $< M, N >$ is called candidate pair.

Besides serving as semantic scanner of clone pairs, the *Action filter* also contributes to making the proposed approach both fast and scalable because it eliminates, early on, candidate pairs for which the likelihood of being clones is low.

Using the notion of method calls to find code similarity has been previously explored by Goffi et al. [49], where method invocation sequences in a code fragment are used to represent a method. We are not interested in the sequence; instead, we use method invocations in a bag of words model, which has been shown to be robust in detecting Type-3 clones [137].

These concepts related to preprocessing, creating two level input partitioning, creating in-memory inverted indexes for fast lookups, and using Action tokens to filter out unlikely clone pairs help us in addressing candidate explosion problems in clone detection. The experiments to demonstrate scalability are presented later in Section 5.4.

4.2.2 Metrics Similarity

Method pairs that survive the size filter and the *Action filter* are passed on to a more detailed analysis of their properties. In the case of Oreo, that detailed analysis focuses on the methods' software metrics. Here we explain the reasons for this decision. The next section dives deeper into the metrics similarity component.

Metrics based approaches for clone detection are known to work very well if the goal is to find only Type-1 and Type-2 clones [81, 105, 120]. This is understandable– given the strict definitions of Type-1 and Type-2, the metric values of such clone pairs should be mostly the same. For Type-3, metrics might look like a good choice too because metrics are resilient to changes in identifiers and literals. However, the use of metrics for clones in the Twilight Zone is not straightforward, because these clones may be syntactically different. As such, the use of metrics requires fine tuning over a large number of configurations between the thresholds of each individual metric. Finding the right balance manually can be hard: for example– is the number of conditionals more meaningful than the number of arguments? To address this issue we use a supervised machine learning approach (Section 4.3).

The method pairs that reach the metrics filter are already known to be similar in size and in their actions. The intuition for using metrics as the final comparison is that methods that are of about the same size and that do similar actions, but have quite different software metrics characteristics are unlikely to be clones.

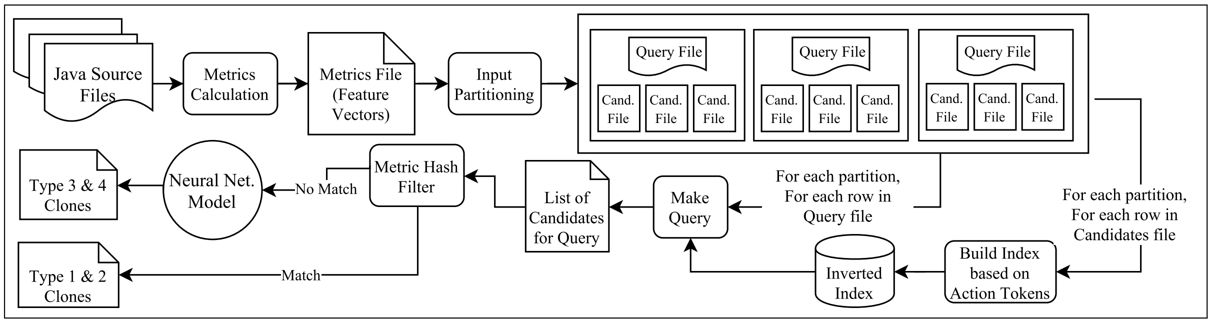


Figure 4.2: Clone Detection Pipeline Process

4.2.3 Clone Detection Pipeline

Figure 4.2 shows Oreo's pipeline in more detail, including the major intermediary data structures. This pipeline makes use of all components presented in this section. Source code files are first given to a *Metrics Calculator* to extract methods and their software metrics. These metrics form the input to Oreo. Then, input partitioning is conducted as described in Section 4.2.1.2, which generates partitions containing query methods and possible candidate methods. Then, for each partition, we create inverted index of its candidates. This inverted index is further partitioned into multiple shards, also explained in Section 4.2.1.2. We then load one of its index-shards into the memory, which is queried with all of the queries in this partition.

For each query method, the index returns a list of candidate clone methods. Then, the hash values of the metrics for each query and its candidates are compared. If metric hash of the query and a candidate is equal, we report them as clones; this is because Type-1 and Type-2 clones have similar structures and thus, equal metric values. If the metric hash is not equal, we pair the candidates with the query and create feature vectors for the candidate pairs. These candidate pairs are then analyzed by the trained model, which predicts if the pair is a clone pair or not. This process is repeated for all partitions and their shards to identify all possible clone pairs. We describe the trained model used in Oreo's pipeline in Section 4.3.2.

4.3 Learning Metrics

For anything other than minor deviations from equality, the use of software metrics for clone detection leads to an explosion of configuration options. To address this issue, we use a supervised machine learning approach. The trained model learns the best configuration of the 24 software metrics from a training set of clones and non-clones. In this section, we describe the dataset used to train the model, and also, the trained model and its selection process.

4.3.1 Dataset Curation

To prepare a dataset, we download 50k random Java projects from GitHub. We then extract methods with 50 or more tokens from these projects; this ensures we do not have empty methods in the dataset. Also, it is the standard minimum clone size for benchmarking [149]. To get *isClone* labels, we used SourcererCC, a state of the art Type-3 clone detector. From this dataset, we randomly sample a labeled dataset of 50M feature vectors, where 25M vectors correspond to clone pairs and other 25M to non clone pairs. Each feature vector has the *isClone* label and 48 metrics (24 for each method).

For model selection purposes, we randomly divide the dataset into 80% pairs for training, and 20% pairs for testing. One million pairs from the training set are kept aside for validation and hyper-parameter tuning purposes. It should be noted that to avoid any significant favorable bias in our experiments, we do not use BigCloneBench's dataset during training.

4.3.2 Deep Learning Model

The model training and model selection were carried out by Prof. Pierre Baldi and his Ph.D. student, Yadong Lu. The dataset used for model training and selection was curated by Farima Farmahinifarahani, a fellow Ph.D. student, and myself.

While there exists many machine learning techniques, here we are using deep learning to detect clone pairs. Neural networks, or deep learning methods are among the most prominent machine learning methods that utilize multiple layers of neurons (units) in a network to achieve automatic learning. Each unit applies a nonlinear transformation to its inputs. These methods provide effective solutions due to their powerful feature learning ability and universal approximation properties. Along with scaling well to large datasets, these approaches can take advantage of well maintained software libraries and can also compute on clusters of CPUs, GPUs and on the cloud. Deep Neural Networks (DNN) have been successfully applied to many areas of science and technology [140], such as computer vision [89], natural language processing [144], and even biology [34].

Here we propose to use a Siamese architecture neural network [9] to detect clone pairs. Siamese architectures are best suited for problems where two objects must be compared in order to assess their similarity, for example comparing fingerprints [9]. Another important characteristic of this architecture is that it can handle the symmetry [109] of its input vector. Which means, presenting the pair $(m1, m2)$ to the model will be the same as presenting the pair $(m2, m1)$. The other benefit brought by Siamese architectures is a reduction in the number of parameters; the weight parameters are shared within two identical sub neural networks making it require fewer number of parameters than a plain architecture with the same number of layers.

Figure 4.3 shows the Siamese architecture model trained for Oreo. Here, the input to the model is a 48 dimensional vector created using the 24 metrics described in Section 4.2.2. This

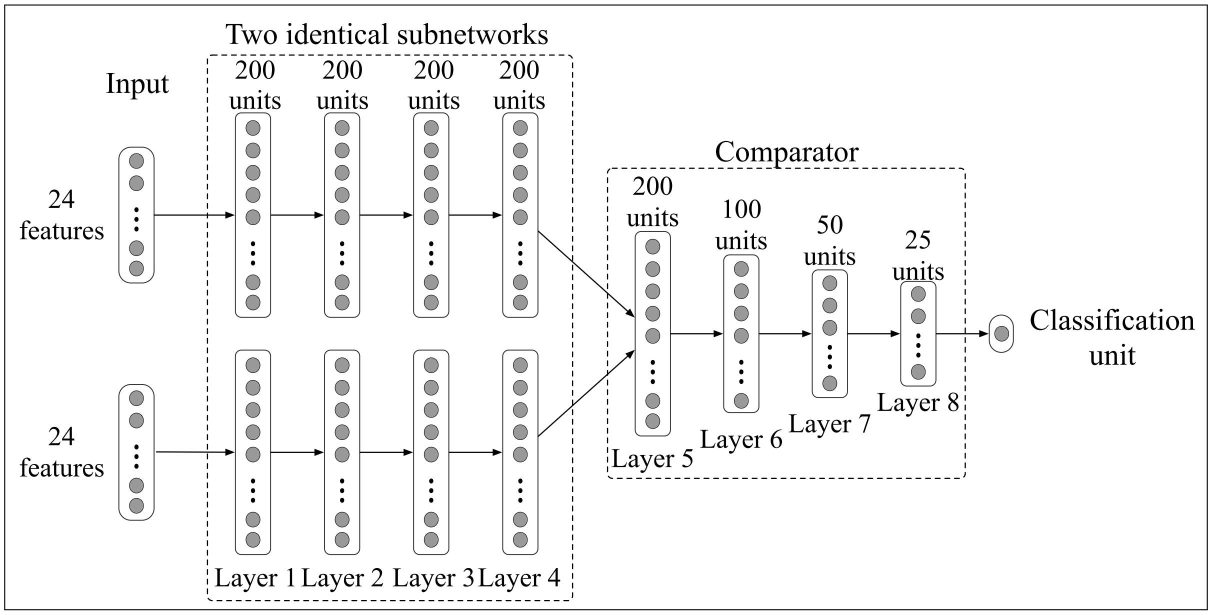


Figure 4.3: Siamese Architecture Model

input vector is split into two input instances corresponding to two feature vectors associated with two methods. The two identical subnetworks then apply the same transformations on both of these input vectors. Both have 4 hidden layers of size 200, with full connectivity (each neuron's output in layer n-1 is the input to neurons in layer n).

The outputs of the two subnetworks are then added and fed to the comparator network which has four layers of sizes 200-100-50-25, with full connectivity between the layers. The output of this comparator network is then fed to the Classification Unit which consists of a logistic unit mathematically represented as $f(\sum_{i=1}^{25} w_i \cdot x_i) = \frac{1}{1+e^{-\sum_{i=1}^{25} w_i \cdot x_i}}$. Where, x_i is the i-th input of the final classification unit, and w_i is the weight parameter corresponding to x_i. The product $w_i \cdot x_i$, is summed over i ranging from 1 to 25 since we have 25 units in Layer 8 (the layer before Classification unit). The output of this unit is a value between 0 and 1, and can be interpreted as the probability of the input pair being a clone. We claim that a clone pair is detected if this value is above 0.5. ($O_i^n = max(I_i^n, 0)$, where O_i^n, I_i^n are respectively the output and input of the i-th neuron in layer n) [45] to produce their output.

In this model, to prevent overfitting, a regularization technique called dropout [10] is applied

Table 4.2: Precision/Recall on Test Dataset

Model	Precision	Recall
Logistic Regression	0.846	0.886
Shallow NN	0.931	0.963
Random forest	0.93	0.94
Plain DNN	0.939	0.972
Siamese DNN	0.958	0.974

to every other layer. In this technique, during training, a proportion of the neurons in a layer is randomly dropped along with their connections with other neurons. In our experiment, we achieve the best performance with 20% dropout. The loss function (function that quantifies the difference between generated output and the correct label) used to penalize the incorrect classification is the relative entropy [91] between the distributions of the predicted output values and the binary target values for each training example. Relative entropy is commonly used to quantify the distance between two distributions. Training is carried out by stochastic gradient descent with the learning rate of 0.0001. The learning rate is reduced by 3% after each training step (epoch), to improve the convergence of learning. The parameters are initialized randomly using 'he normal'[55], a common initialization technique in deep learning. Training is done in minibatches where the parameters are updated after training on each minibatch. Since the training set is large, we use a relative large minibatch size of 1,000.

4.3.3 Model Selection

To find the model, we experiment with several architectures, for each architecture, several number of layers and units, and several hyper-parameter settings such as learning rate (the rate for updating the weight parameters) and loss function. To compare these models, we compute several classification metrics including accuracy (the rate of correct predictions based on validation dataset labels), Precision (the fraction of retrieved instances that are

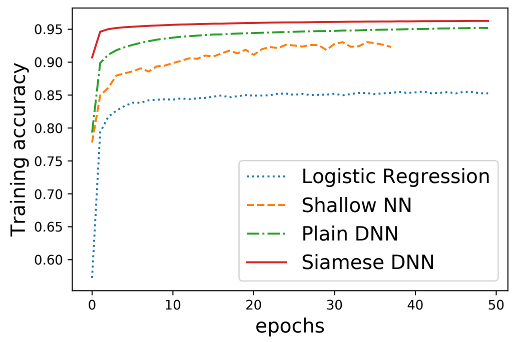


Figure 4.4: Training Accuracy

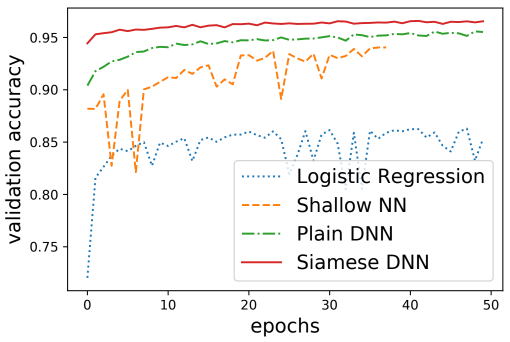


Figure 4.5: Validation Accuracy

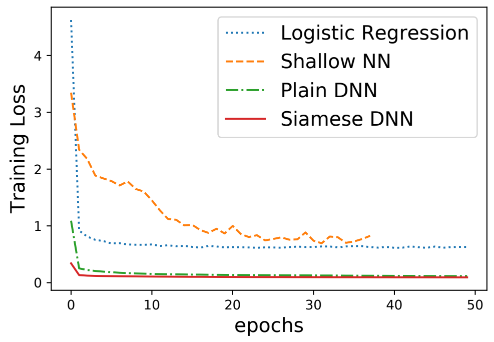


Figure 4.6: Training Loss

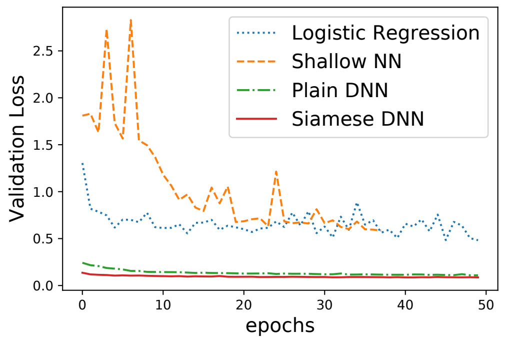


Figure 4.7: Validation Loss

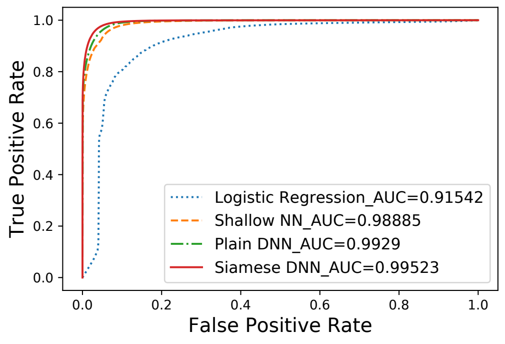


Figure 4.8: ROC Plot and AUC Values

relevant), Recall (the fraction of relevant instances that are retrieved), Receiver Operating Characteristic (ROC) curve (true positive rate against false positive rate), and Area Under the ROC Curve (AUC). The selection process is described in the rest of this section.

As mentioned, in the process of selecting the best model, we also train other models based on different architectures including: (1) A simple logistic regression model, (2) A shallow neural network (Shallow NN) model with a single hidden layer and similar amount of parameters as in Siamese model, and (3) a plain fully connected network (Plain DNN) with the same layer sizes as the full Siamese architecture. For each architecture, we train many models; and for the sake of simplicity, here we compare the best model from each mentioned architecture. All models are trained on the same dataset for 50 epochs and training is terminated if the validation loss stops increasing for two consecutive epochs.

Results comparing the best model from each mentioned architecture are reported in Figure 4.4 to Figure 4.8, as well as in Table 4.2. The Siamese network model outperforms all other models in every metric. Figure 4.4 illustrates the accuracy attained by each model through the epochs in Training, and Figure 4.5 shows the same concept in Validation. The Siamese Deep Neural Network and Plain DNN have better accuracy than other two models. However, the Siamese DNN, designed to accommodate the symmetry property of its overall input, outperforms the accuracy of the Plain DNN. More importantly, this model is performing better than the Plain DNN on the validation set, despite using significantly less free parameters. Thus, the Siamese architecture is considered to have better generalization properties on new samples. Figure 4.6 depicts the decrease in training loss over the epochs for each model, and Figure 4.7 shows the same concept for validation loss. In Figure 4.6, we observe that the training loss for logistic regression and shallow NN models stops improving at around 0.8. Whereas, the loss for plain NN and Siamese DNN can go below 0.09 as we train longer. A similar pattern is observed for validation loss in Figure 4.7. The large fluctuations for shallow NN are due to the small size of the validation set.

Figure 4.8 shows the ROC curves of the different classifiers and compares the corresponding AUC values for validation dataset. A high AUC value denotes a high true positive rate and low false negative rate. As shown in Figure 4.8, the Siamese architecture leads to the best AUC value (0.995). Finally, Precision and Recall performances on test dataset are compared in Table 4.2. The table shows that the Siamese DNN has a recall comparable with the Plain DNN, but has a better precision (0.958 vs 0.939). Totally, Siamese DNN outperforms other models in both precision and recall values.

Other than the mentioned differences, compared to the plain network, the Siamese architecture model has around 25,000 parameters which is 37% less than the plain structure, which leads to less training time and less computation burden.

4.4 Related Work

State-of-the-art up to Type-3 clones. Currently, NiCad [127], SourcererCC [137], and CloneWorks [153] are the state of the art in detecting up to Type-3 clones. While both SourcererCC and CloneWorks use hybrid of Token and Index based techniques, NiCad uses a text based approach involving syntactic pretty-printing with flexible code normalization and filtering. Deckard [64] builds characteristic vectors to approximate the structural information in source code's AST, and then clusters these vectors. We compare Oreo with these approaches in Section 5.4.

Techniques to detect Type-4 clones. Gabel et al. [43] find semantic clones by augmenting Deckard [64] with a step for generating vectors for semantic clones. Jiang et al. [65] proposed a method to detect semantic clones by executing code fragments against random inputs. Both of these techniques have been implemented to detect C clones. Unfortunately, precision and recall are not reported, so we cannot compare.

Machine learning techniques. White et al. [159] present an unsupervised deep learning approach to detect clones at method and file levels. They explored the feasibility of their technique by automatically learning discriminating features of source code. They report the precision of their approach to be 93% on 398 files and 480 method-level pairs across eight Java systems. However, recall is not reported on a standard benchmark like BigCloneBench, and there is no analysis of scalability.[2]

In Wei and Li's approach [158], a Long-Short-Term-Memory network is applied to learn representations of code fragments and also to learn the parameters of a hash function such that the Hamming distance between the hash codes of clone pairs is very small. They claim that they tested their approach on BigCloneBench and OJClone datasets. However, it is unclear which dataset they have used for training, and their technique's scalability is not reported.[3]

Sheneamer and Kalita use AST and PDG based techniques to generate semantic and syntactic features to train their model [141]. Their approach differs from ours in two ways: i) the features they use are different from ours and we do not use semantic features; instead, we use a semantic filter and ii) they use simple and ensemble methods to train their model, whereas we use deep learning. Also, the dataset they use to train and test their model is unclear.[4]

CCLearner [92] is another approach that leverages deep learning to classify clone and non-clone method pairs. In this work, authors have used the information about the usage of terms in source code (such as reserved words or identifiers) to build the feature vector for each method. They have used a part of BigCloneBench for training the classifier, and the rest

[2]We contacted the authors in order to get their tool. They provided us with their source code but it needed a considerable amount of effort to be used. We were also not provided with their train dataset or their trained model. Under these circumstances, we decided not to pursue a comparison of their approach with ours.

[3]We contacted the authors to get their tool, but failed to get any response.

[4]Here, too, we contacted the authors but did not receive any response.

for testing it. However, we reserved the whole BigCloneBench dataset for testing purposes and trained our model on a different dataset in order to avoid any biases or getting overfitted to test dataset. They have tested the scalability of their approach on a dataset of 3.6 MLOC, whereas, we tested Oreo on the entire IJaDataset which is 250 MLOC of size.

4.5 Chapter Summary

In this chapter, we introduced a novel approach for code clone detection. Oreo is a combination of information-retrieval, machine-learning, and metric-based approaches. We introduced a novel Action Filter and a two-level input partitioning strategy, which reduces the number of candidates while maintaining good recall. We also introduced a deep neural network with Siamese architecture, which can handle the symmetry of its input vector; A desired characteristic for clone detection.

Chapter 5

Evaluation of Oreo

We compare Oreo's detection performance against the latest versions of the five publicly available clone detection tools, namely: SourcererCC [137], NiCad [127], CloneWorks [153], Deckard [64], and CCAligner [156].

We also wanted to include tools such as SeByte [77], Kamino [116], JSCTracker [40], Agec [71], and approaches presented in [141, 154, 158, 159], which claim to detect Type-4 clones. On approaching the authors of these tools, we were communicated that the tool implementation of their techniques currently does not exist and with some authors, we failed to receive a response. Authors of [43] and [65] said that they do not have implementations for detecting Java clones (They work either on C or C++ clones).

As Type-1 and Type-2 clones are relatively easy to detect, we focus primarily on Type-3 clone detectors. The configurations of these tools, shown in Table 5.1, are based on our discussions with their developers, and also on the configurations suggested in [149]. For Oreo, we carried out a sensitivity analysis of *Action filter* threshold ranging from 50% to 100% at a step interval of 5%. We observed a good balance between recall and precision at the 55% threshold.

Table 5.1: Tools' Approach and Configurations

Tool Name	Approach	Configuration
CloneWorks (A)	Token and Index based	Min tokens=1, Similarity threshold = 70%, Mode=Aggressive
CloneWorks (C)	Token and Index based	Min tokens=1, Similarity threshold = 70%, Mode=Conservative
CCAligner	Token based	Min lines=6, Similarity threshold = 60%, Edit distance=1, Window size=6
Deckard	Tree based	Min tokens=50, Stride=2, Similarity threshold= 85%
NiCad	Text based	Min lines=6, Blind identifier normalization =True, Literal abstraction=True, Difference threshold= 30%
Oreo	Metrics and Machine learning based	Min tokens=15, Action filter threshold= 55%, Input partition threshold= 60%
SourcererCC	Token and Index based	Min tokens=1, Similarity threshold= 70%

The remainder of this chapter is organized as follows. Section 5.1 elaborates on the experiments conducted to compare Oreo's recall with other tools. Similarly, Section 5.2 and Section 4.2.1 compares the precision and scalability of Oreo to other tools. We present the manual analysis of clone pairs in Section 5.5; Section 5.6 presents the limitations of this study, and finally, we conclude the chapter in Section 5.7.

5.1 Recall

The recall of these tools is measured using Big-CloneEval [150], which performs clone detection tool evaluation experiments using BigCloneBench [149], a benchmark of real clones. Big-CloneEval reports recall numbers for Type-1 (T1), Type-2 (T2), Type-3, and Type-4 clones. For this experiment, we consider all clones in BigCloneBench that are 6 lines and 50 tokens in length or greater. This is the standard minimum clone size for measuring recall [19, 149].

To report numbers for Type-3 and Type-4 clones, the tool further categorizes these types into four subcategories based on the syntactical similarity of the members in the clone pairs, as follows: i) Very Strongly Type- 3 (VST3), where the similarity is between 90-100%, ii) Strongly Type-3 (ST3), where the similarity is between 70-90%, iii) Moderately Type-3

Table 5.2: Recall and Precision Measurements on BigCloneBench

Tool	Recall Results												Precision Results
	T1 (35,802)		T2 (4,577)		VST3 (4,156)		ST3 (15,031)		MT3 (80,023)		WT3/T4 (7,804,868)		
	%	#	%	#	%	#	%	#	%	#	%	#	Sample Strength=400
Oreo	**100**	**35,798**	**99**	**4,547**	**100**	**4,139**	89	13,391	**30**	**23,834**	0.7	57,273	82.5%
SourcererCC	100	35,797	97	4,462	93	3,871	60	9,099	5	4,187	0	2,005	**98.5**%
CloneWorks (A)	100	35,777	99	4,544	98	4,090	**93**	**13,976**	3	2,700	0	35	77.25%
CloneWorks (C)	100	35,750	97	4,435	92	3,803	60	9,020	5	4,070	0	1,834	95.75%
CCAligner	100	35,724	99	4,526	97	4,048	70	10,483	10	8,023	0	12,540	71.25%
NiCad	100	35,769	99	4,541	98	4,091	93	13,910	0.8	671	0	12	94%
Deckard*	60	21,481	58	2,655	62	2,577	31	4,660	12	9,603	**1**	**780,487**	34.8%

* Absolute numbers for Deckard are calculated based on the reported percentage values

(MT3), where the similarity is between 50-70%, and iv) Weakly Type-3/Type-4 (WT3/4), where the similarity is between 0-50%. Syntactical similarity is measured by line and by language token after Type-1 and Type2 normalizations.

Table 5.2 summarizes the recall number for all tools. The recall numbers are summarized per clone category. The numbers in the parenthesis next to the category titles show the number of manually tagged clone pairs for that category in the benchmark dataset. Each clone category has two columns under it, titled "%", where we show the recall percentage and "#", where we show the number of manually tagged clones detected for that category by each tool. The best recall numbers are presented in *bold typeface*. We note that we couldn't run Deckard on the BigCloneEval as Deckard produced more than 400G of clone pairs and BigCloneEval failed to process this huge amount of data. The recall numbers shown for Deckard are taken from SourcererCC's paper [137], where the authors evaluated Deckard's recall on BigCloneBench. The total number of clone pairs are not available for Deckard, and for this reason, we calculated them based on the reported percentage values.

As Table 5.2 shows, Oreo performs better than every other tool on most of the clone categories, except for ST3 and WT3/T4. CloneWorks performs the best on ST3 and Deckard performs the best on WT3/T4. Performance of Oreo is significantly better than other tools on the harder-to-detect clone categories like MT3 and WT3/T4, where Oreo detects one to two orders of magnitude more clone pairs than SourcererCC, CloneWorks, and NiCad. This is expected as these tools are not designed to detect harder-to-get clones in the Twilight Zone. In the ST3 category SourcererCC's recall (60%) is significantly lower than CloneWorks (A) (93%) and NiCad (93%). This could explain why Oreo, which is trained using SourcererCC, did not perform as well as CloneWorks and NiCad in this category. SourcererCC filters out many pairs when it cannot find enough shared tokens in them. Since in harder-to-detect clone categories the overlap similarity in tokens is low, SourcererCC eliminates many pairs in these categories. However, software metrics, used by Oreo are resilient to changes in identifiers and literals which makes Oreo perform much better than SourcererCC in the Twilight Zone.

The recall numbers are encouraging as they show that besides detecting easier to find clones such as T1, T2, and VST3, Oreo detects clones that are hardly detected by other tools. Compared to the other tools, where the maximum recall is 12% by Deckard, 30% of recall in MT3 category is a great improvement. Given that SourcererCC has only 5% recall in MT3 category and 60% recall in ST3, we believe recall of Oreo can be increased further by training the DNN model with more samples of MT3 and ST3 categories.

5.2 Precision

In the absence of any standard benchmark or tool we compare precision of these tools manually – a common practice to measure precision of clone detectors [137].

Methodology. For each tool we randomly selected 400 clone pairs, a statistically significant sample with 95% confidence level and 5% confidence interval, from the clone pairs detected by each tool in the recall experiment. The validation of clones were done by three judges. The judges were kept blind from the source of each clone pair. Unlike many classification tasks that require fuzzy human judgment, this task required following very strict definitions of what constitutes Type-1, Type-2, Type-3, and Type-4 clones. We note that all three judges are experts in software clones and are aware of clone and clone types definitions. We aggregated the votes by taking majority vote.

Table 5.2 shows precision results for all tools. We found that the precision of Oreo is 82.5%. Oreo performed better than Deckard, CCAligner and CloneWorks (A) performed better than Oreo. Deckard's precision is the lowest at 34.8% and SourcererCC's precision is the highest at 98.5%. While the precision of Oreo is lower than the other three state of the art tools, it is important to note that Oreo pushes the boundaries of clone detection to the categories where other tools have almost negligible performance.

5.3 Precision in the Twilight Zone

The precision experiment results presented earlier give an overall view on the precision of the tools, however, they miss out on one important information: if a tool's precision is low/high, on which type of clones is its precision low/high? If a tool's precision is low/high, does it mean that its precision is low/high on all types of clones? To find answers to these questions, we classify the clone pairs reported by each tool into Type I, Type II, and Type III clone types. Section 5.3.1 elaborates on how we categorized the pairs into different types.

Table 5.3 presents the results of this classification exercise. The first column of this table shows tools' names, and the second column shows the total number of clone pairs detected by each tool. Some of the clone pairs reported by the tools included code fragments that were not parseable; hence, we removed those pairs from our study. The third column shows the number of pairs that were parseable and were used in our experiments. Then, for each clone type, there are two columns: the column with # header shows the total number of pairs in the corresponding clone type, and the column with %, shows the percentage of clone pairs categorized in that type.

Table 5.3: Number of Clone Pairs Reported by Each Tool Per Type

Tool Name	Total	Total Parsed	Type I		Type II		Type III	
			#	%	#	%	#	%
Oreo	4,186,474	4,186,474	942,078	22%	198,691	5%	3,045,705	73%
SourcererCC	15,689,823	15,689,823	1,835,510	12%	8,317,571	53%	5,536,742	35%
CloneWorks (A)	653,053,676	651,739,063	1,902,706	0.3%	576,764,902	88.5%	73,071,455	11.2%
CloneWorks (C)	13,296,023	13,286,980	1,873,445	14%	7,364,762	55%	4,048,773	31%
CCAligner	4,253,798	4,252,328	938,421	22%	178,766	4%	3,135,141	74%
NiCad	7,144,918	7,138,729	1,516,232	21%	317,371	5%	5,305,126	74%

As we can see from the table, these tools detect different types of clones in different proportions and these proportions differ for all tools. For example, the proportion of Type I clones pairs reported by all tools vary between 0.3% (*CloneWorks (A)*) and 22% (*Oreo* and *CCAligner*). In studying the Type II pairs statistics, we see that a majority of the pairs reported by Aggressive mode of CloneWorks are of Type II (88.5%). About half of the pairs reported by SourcererCC (53%) and CloneWorks(C) (55%) are categorized into Type II category. For other tools, most of their predicted pairs are in Type III category: the percentage of Type III pairs in NiCad is the greatest among others (74%), and then the largest numbers are CCAligner with 74% and Oreo with 73%.

Type Based Experiment Setup. The setup is similar to the setup of earlier precision experiments. With this difference that after separating each tool's clone pairs by type, we made 400 samples from each tool, each type. Then, we aggregated and shuffled the type-based samples of each tool, and showed a set of 1200 pairs (400 pairs from each clone type) for each tool to the same three judges. The judges were kept blind from the source and the type of the clone pairs to remove any bias towards knowing the tools or clone types.

Results. The results of these experiments are presented in the columns under *Type-based* in Table 5.4. For the sake of comparison, the All-types precision numbers are also presented.

The type-based results show that all tools performed near perfect on Type I and Type II

Table 5.4: Type-Based Precision

Tool Name	All-types (400 Pairs)	Type-based (1200 Pairs Totally)		
		Type I (400 Pairs)	Type II (400 Pairs)	Type III (400 Pairs)
Oreo	82.5%	100%	99.75%	88%
SourcererCC	98.5%	100%	100%	98.5%
CloneWorks (A)	77.25%	100%	99.25%	74.75%
CloneWorks (C)	95.75%	100%	100%	99.5%
CCAligner	71.25%	100%	100%	67.75%
NiCad	94%	100%	99.25%	93%

categories. The difference is mostly in the precision of Type III clone category. For example it can be observed for CCAligner, Oreo, and CloneWorks (A) that their *All-types* precision is not perfect; however, the type-based experiment shows that they both have perfect (or near perfect) precision in Type I and Type II categories, and the category that lowers their precision is Type III. Also, Oreo and CCAligner focus on retrieving the harder to get Type III clones (mostly MT3 subcategory), which are more error prone than detecting the earlier Type III subcategories. This explains why the precision numbers for Oreo and CCAligner lower than the others.

To investigate deeper, we further classified the Type III clone pairs of each tool into different Type III subcategories. We then performed precision experiment for all tools considering only the pairs which fall into the Twilight Zone. The process of classification of Type III clones into different Type IIIsubcategories is explained in Section 5.3.1.

Twilight Zone Precision Experiment Setup. The setup is similar to the setup of earlier precision experiments. With this difference that after separating each tool's Type III clone pairs, we combined the pairs in the *MT3* and *WT3* to get the clone pairs in the Twilight Zone. From the Twilight Zone pool of clone pairs we made 400 samples from each tool. Then, we aggregated and shuffled the type-based samples of each tool and showed them to the same three judges. The judges were kept blind from the source of the clone pairs.

Results. The results of these experiments are presented in the columns under *Type-based* in Table 5.5. For the sake of comparison, the All-types precision numbers are also presented. CloneWorks(C) and SourcererCC perform very well with precision greater than 90%. Oreo's precision is better in the Twilight Zone than that of NiCad (64%), CCAligner (59%), and CloneWorks(A) (40%).

Another metric to measure the effectiveness of clone detectors tools is F1 measure. F1 measure (scaled to the range 0 - 100), shown in equation 5.1, for a clone detector is calculated by taking a harmonic mean of its recall and precision values.

$$F1 = \frac{2 * R * P * 100}{R + P} \tag{5.1}$$

The recall values for all clone detectors is close to zero, and therefore the F1 measure for all clone detectors is almost zero. Oreo's F1 measure in the Twilight Zone, however, is 2.0.

We decided to compute the F1 value for each clone detector for the MT3 category, as their recall values are not zero in this category. To this end, we conducted another set of precision experiments to estimate the precision of each tool in the MT3 category. This set of experiments is also conducted with the same set of three judges who were kept blind from the source of clone pairs. Column *MT3* (fifth column) in Table 5.5 shows the precision values of each tool in MT3 category. Column *F1 for MT3* (sixth column) shows the F1 measure computed for all tools in MT3 category. Oreo Performs the best with F1 score of 41. CCAligner performs second best with F1 score of 17. Rest all other tools have F1 score below 10. Please see Table 5.6 for recall and precision results in the MT3 category.

Table 5.5: Type-Based Performance

Tool Name	All-types	Type III	Twilight Zone	MT3	F1 for MT3
Oreo	82.5%	88%	80%	69%	41.8
SourcererCC	98.5%	98.5%	94%	93%	9.49
CloneWorks (A)	77.25%	74.75%	40%	44%	5.61
CloneWorks (C)	95.75%	99.5%	96%	96%	9.50
CCAligner	71.25%	67.75%	59%	69%	17.5
NiCad	94%	93%	64%	62%	1.58

Table 5.6: MT3 Performance

Tool Name	Recall MT3	Precision MT3	F1 MT3
Oreo	30%	69%	41.8
SourcererCC	5%	93%	9.49
CloneWorks (A)	3%	44%	5.61
CloneWorks (C)	5%	96%	9.50
CCAligner	10%	69%	17.5
NiCad	0.8%	62%	1.58

The recall and precision experiments demonstrate that among the clone detectors used in this study, Oreo has the best detection performance in the Twilight Zone.

5.3.1 Classification of Clone Pairs

Most tools do not report clone type information along with their clone pairs. Therefore, we devised algorithms to classify the code transformations that underly the definitions of different types of clones. Using these algorithms, we first identify Type I pairs in the reported clone pairs of a tool. The identified Type I pairs are then kept aside. Then, from the remaining pairs, we identify Type II pairs and keep them aside. The remaining pairs are then tagged as Type III pairs. To further classify pairs Type III pairs into different subcategories (VST3, ST3, MT3, and WT3), we devised algorithms following the definitions of these subcategories provided by Svajlenko and Roy [148].

It should be noted that with these algorithms we were not trying to identify a pair as clone or not. These algorithms are simply classifying pairs based on clone type and subcategory definitions. For instance, the two code fragments illustrated in Listing 5.1 fit the Type II definition: they are identical code fragments except for variations in identifier names and literal values. However, a human judge may not classify them as a true clone pair because she may argue that they are not performing the same functionality. If they were clones, they would be Type II; but they are not.

Algorithm 1 Algorithmic implementation of Type I classification

INPUT: $C1$ and $C2$ are two string variables storing the bodies of two code fragments. **OUTPUT:** *Boolean*

```
1: function isTypeOne(C1, C2)
2:     C1 = RemoveComments(C1)
3:     C2 = RemoveComments(C2)
4:     C1 = RemoveWhitespacesAndNewLines(C1)
5:     C2 = RemoveWhitespacesAndNewLines(C2)
6:     return MD5(C1)==MD5(C2)
7: end function
```

We discuss the algorithms in the rest of this section.

Listing 5.1: Example of a Type II False Positive Pair

```
public GBrowseGFFBuilder(String build, String dbHost, String dbUser, String dbPassword, String dbPort, String
        outputFile) {
    this.build = build;
    this.dbHost = dbHost;
    this.dbUser = dbUser;
    this.outputFile = outputFile;
}
--------------------------------
public TopicProfile(String id, String name, String siteid, String sitename, String unitid, String unitname) {
    this.id = id;
    this.name = name;
    this.siteid = siteid;
    this.unitname = unitname;
}
```

Type I Transformations. Type I definition implies that a clone pair can be classified as a Type I if the two code fragments are exact replicas, except for differences in white spaces, comments, and layout. Hence, in order to identify Type I transformations, we parsed each code fragment and removed the white spaces and comments from it [1]. Then we treated the resulting fragment as a string, and calculated the MD-5 hash code of this string, which we call Type I hash. If the Type I hash of two code fragments in a reported clone pair are equal, this pair is marked as a Type I pair. The pseudocode for implementation of these rules is shown in Algorithm 1.

Type II Transformations. Type II definition indicates that two code fragments are classified as Type II if, in addition to Type I differences, they only differ in identifier names and literal values. Hence, we implemented code normalizations on identifiers and literals to identify Type II transformations. That is, after removing the set of Type I pairs from the clone pairs reported by each tool, we applied the normalizations to code fragments of each clone pair. These normalizations include replacing all identifiers with a fixed value, and then, replacing all literal values with fixed values according to literal types (String, Character, Boolean, Integer, Long, Float, Double). This process removes the differences pertaining to Type II cloning, and hence, resulting normalized code fragments can be compared using Type I rules: if two normalized code fragments in a clone pair satisfy Type I transformation

[1]Since Java syntax is not sensitive to layout, we ignored layout differences.

rules, they are categorized as Type II pairs. The pseudocode for implementing these rules is shown in Algorithm 2.

Algorithm 2 Algorithmic implementation of Type II classification

INPUT: $C1$ and $C2$ are two string variables storing the bodies of two code fragments. **OUTPUT:** *Boolean*

```
1: function isTypeTwo(C1, C2)
2:     C1 = ReplaceIdentifiers(C1)
3:     C2 = ReplaceIdentifiers(C2)
4:     C1 = ReplaceLiterals(C1)
5:     C2 = ReplaceLiterals(C2)
6:     return isTypeOne(C1,C2)
7: end function
```

Type III Subcategory Classification.

After classifying clone pairs into Type I, Type II, and Type III categories, we further classify the Type III clone pairs into different Type III subcategories. We use Algorithm 3 to achieve this classification. In this algorithm, we first removed any code comments from the method bodies and then performed Type II normalizations. Then, we count the common tokens in the normalized method bodies and divide this count with the tokens count of the bigger method to give us a similarity score between two methods. Using this similarity score we classify the pair into the subcategories by following the definitions of VST3, ST3, MT3, and WT3 [148]. The pairs in MT3 and WT3 are then combined to get pairs in the Twilight Zone.

Algorithm 3 Algorithm to classify clone pairs into Type III subcategories

INPUT: $C1$ and $C2$ are two string variables storing the bodies of two code fragments. **OUTPUT:** *Boolean*

```
1: function classifyIntoType3Subcategories(C1, C2)
2:     C1 = RemoveComments(C1)
3:     C2 = RemoveComments(C2)
4:     C1 = ReplaceIdentifiers(C1)
5:     C2 = ReplaceIdentifiers(C2)
6:     C1 = ReplaceLiterals(C1)
7:     C2 = ReplaceLiterals(C2)
8:     CommonTokensCount = CountCommonTokens(C1, C2)
9:     MaxTokensCount = GetMaxTokensIn(C1, C2)
10:    Similarity = GetSimilarity(CommonTokensCount, MaxTokensCount)
11:    Classify(C1,C2, Similarity)
12: end function
```

5.4 Scalability

As mentioned before, scalability is an important requirement for the design of Oreo. Most metrics-based clone detectors, including the recent machine learning based ones, tend to grow quadratically with the size of input, which greatly limits the size of dataset to which they can be applied.

We demonstrate the scalability of Oreo in two parts. In the first part, we show the impact of the two-level input partitioning and Action Filter on the number of candidates to be processed. As a reminder, reducing the number of candidates early on in the pipeline, greatly improves the scalability of clone detection tools. The second part is a run time performance experiment.

Dataset for scalability experiments: We use the entire IJaDataset [4], a large inter-project Java repository containing 25,000 open-source projects (3 million source files, 250MLOC) mined from SourceForge and Google Code. Only 2 other tools (SourcererCC and CloneWorks) have been shown to scale to this dataset.

Number of candidates. To measure the impact of the Action Filter and two-level input partitioning on the number of candidates, we selected 1,000 random methods as queries from the dataset. We then executed these 1,000 queries on our system to see the impact of the Action filter and input partitioning on the number of candidates to be sent to the metrics-based DNN model. The threshold for Action filter was set to 55%. Also we selected 6 as the number of partitions for input partitioning.

Table 5.7 summarizes the impact. The top row shows the base line case where each query is paired with every method in the dataset, except for itself. This is the *modus operandi* of many clone detection tools. In the second row, the Action-filter's similarity was set to 1% to minimize it's impact, however, partitions were turned on. For the results in third

Table 5.7: Impact of Action Filter and Input Partitioning

Action Filter	Input Partitioning	Num-Candidates
No Filter	No partitions	2,398,525,500
1%	on	58,121,814
55%	No partitions	260,655
55%	on	218,948

row, we had kept the Action filter on at 55% similarity threshold but we switched off the partitioning. The bottom row shows the results for number of candidates when both Action filter and input partitioning were switched on. We can see that Action filter has a strong impact on reducing the number of candidate pairs. The partitioning lets us do in-memory computations; together they both help us achieve high speed and scalability.

Run time and resource demands. We used an Intel(R) Xenon(R) CPU E5-4650 2.20GHz machine with 112 cores, 256G memory, and 500G of solid state disk. We modified the tool configurations to detect clones >=10 source lines of code and we also limited the memory usage of each tool to 12G and disk usage to 100G to simulate the scale experiment conditions as described in [137]. We did not run this experiment for NiCad and Deckard as they were previously shown to not scale to this input at the given experiment settings.

Oreo scaled to this input, taking 26 hours and 46 minutes, SourcererCC took 4 hours and 40 minutes, and CloneWorks took 1 hour and 50 minutes. Oreo took more time than both SourcererCC and CloneWorks. It is worth noting that SourcererCC and CloneWorks validate candidates in the Type-1, Type-2 and early categories of Type-3 before reporting them as clones. They use aggressive heuristics to filter out pairs which are beyond ST3 category. Oreo, on the other hand, evaluates pairs in the Twilight Zone as well, which increases the number of candidates to be evaluated by a non trivial number. Unfortunately, we do not have the number of candidate pairs validated by each tool. Nevertheless, this experiment demonstrates that Oreo scales to large datasets in a reasonable time.

The scalability experiment along with recall and precision experiments, demonstrates that Oreo is a scalable clone detector, capable of detecting not just easy categories of clone, but also in the Twilight Zone with high correctness. In literature we did not find any other software metrics based clone detector which is scalable to such large datasets. Also, as demonstrated by the experiments, Oreo's performance in the Twilight Zone is better than the other existing tools. At this point I can reiterate my thesis:

Oreo improves the state of the art in code clone detection by being the most scalable and the most effective technique known so far to detect clones in the Twilight Zone.

5.5 Manual Analysis of Semantic Clones

During the precision study, we saw pairs which were hard to classify into a specific class. We also observed some examples where the code snippets had high syntactic similarity but semantically they were implementing different functionality and vice-versa.

We saw an interesting pair, in which one method's identifiers were written in Spanish and the other's in English. These methods offered similar, but not exactly the same, functionality of copying content of a file into another file. The method written in English iterated on a collection of files and copied the contents of each file to an output stream. The method in Spanish copied the content of one file to an output stream. Action Filter correctly identifies the semantic similarities in terms of the library calls of these methods, and later our DNN model correctly recognizes the structural similarity, ignoring the language differences.

Here, we present two examples of clone pairs with high semantic similarity and low syntactic similarity. Listing 5.2 shows one of the classical examples of Type-4 clone pairs reported by Oreo. As it can be observed, both of these methods aim to do sorting. The first one implements *Insertion Sort*, and the second one implements *Bubble Sort*. The *Action filter*

Listing 5.2: Clone Pair Example: 1

```java
private void sortByName() {
    int i, j;
    String v;
    for (i = 0; i < count; i++) {
        ChannelItem ch = chans[i];
        v = ch.getTag();
        j = i;
        while ((j > 0) && (collator.compare(chans[j - 1].getTag(), v) > 0)) {
            chans[j] = chans[j - 1];
            j--;
        }
        chans[j] = ch;
    }
}
----------------------------------------
public void bubblesort(String filenames[]) {
    for (int i = filenames.length - 1; i > 0; i--) {
        for (int j = 0; j < i; j++) {
            String temp;
            if (filenames[j].compareTo(filenames[j + 1]) > 0) {
                temp = filenames[j];
                filenames[j] = filenames[j + 1];
                filenames[j + 1] = temp;
            }
        }
    }
}
```

Listing 5.3: Clone Pair Example: 2

```java
public static String getExtension(final String filename) {
    if (filename == null || filename.trim().length() == 0 || !filename.contains(".")) return null;
    int pos = filename.lastIndexOf(".");
    return filename.substring(pos + 1);
}
----------------------------------------
private static String getFormatByName(String name) {
    if (name != null) {
        final int j = name.lastIndexOf('.') + 1, k = name.lastIndexOf('/') + 1;
        if (j > k && j < name.length()) return name.substring(j);
    }
    return null;
}
```

Listing 5.4: False Positive Example

```java
public static String getHexString(byte[] bytes) {
    if (bytes == null) return null;
    StringBuilder hex = new StringBuilder(2 * bytes.length);
    for (byte b : bytes) {
        hex.append(HEX_CHARS[(b & 0xF0) >> 4]).append(HEX_CHARS[(b & 0x0F)]);
    }
    return hex.toString();
}
----------------------------------------
String sequenceUsingFor(int start, int stop) {
    StringBuilder builder = new StringBuilder();
    for (int i = start; i <= stop; i++) {
        if (i > start) builder.append(',');
        builder.append(i);
    }
    return builder.toString();
}
```

finds many common *Action tokens* like three instances of *ArrayAccess* action tokens, and 2 instances of *ArrayAccessBinary* action tokens, leading to a high semantic match. Further, the trained model finds high structural match as both models have two loops where one is nested inside another; first method declares three variables whereas the second declares four. Oreo does not know that both functions are implementing different sorting algorithms, and hence catching a Type-4 clone here can be attributed to chance. Nevertheless, these two implementations share enough semantic and structural similarities to be classified as a clone pair by Oreo.

Another example is illustrated in Listing 5.3 where both methods attempt to extract the extension of a file name passed to them. The functionality implemented by both methods is the same, however, the second method does an extra check for the presence of / in its input string (line 9). We were unsure whether to classify this example as a WT3/T4 or an MT3 since, although some statements are common in both, they are placed in different positions. Moreover, the syntactic similarity of tokens is also low as both methods are using different variable names. These examples demonstrate that Oreo is capable of detecting semantically similar clone pairs that share little syntactical information.

Besides true positives, we found some false positives too. An example is shown in Listing 5.4. *Action filter* captures similar occurrences of *toString()* and *append()* in both methods and finds a high semantic match. The DNN model also finds the structures of both of these methods to be similar as both contain a *loop*, an *if statement*, and both declare same number of variables, leading to the false prediction. Having a list of stop words for *Action tokens* repeated in many code fragments may help filter out such methods.

5.6 Limitations of this Study

The training and evaluation of models is done only on Java methods. Adaptation to other languages is possible, but requires careful consideration of the heuristics and the software metrics described here. The *Action filter* we propose may not work for small methods, that are very simple and neither make a call to other methods nor do they refer to any class properties. In this study, the minimum threshold of 50 tokens removes the simpler methods, making *Action filter* work well. If we decide to pursue clone detection in small methods, we will explore the option of adding method names or their derivatives to mitigate this concern. The clone detection studies are affected by the configuration of the tools [157]. We mitigate this risk by contacting the authors of different tools and using the configurations suggested by them. The precision study could be affected by human bias. We mitigate it by involving two judges. This bias, however, can be further reduced by involving more judges.

5.7 Chapter Summary

The chapter compares Oreo with four other state of the art tools on a standard benchmark and demonstrates that Oreo is scalable, accurate, and it significantly pushes the boundaries of clone detection to harder-to-detect clones in the Twilight Zone. In future, we will explore the possibilities and impacts of training more models at finer granularities, and training using the clones detected by an ensemble of clone detection tools to improve both the recall and the precision of harder-to-detect semantic clones.

Chapter 6

InspectorClone: Precision Studies Made Easy

6.1 Introduction

The effectiveness of clone detection tools is usually evaluated in terms of precision and recall. The measurement of precision and recall, in general, relies on the existence of labeled datasets. A good labeled dataset provides realistic data and credible labels on all the constituents that should be detected by the analysis tool – in the case of clone detection, all clone pairs are labeled as such. A labeled dataset for clones allows one to measure how many of the clones identified by a certain tool are indeed clones or not (precision), and how many of the true clone pairs are detected by the tool (recall). Publicly available labeled datasets, also known as benchmarks, allow direct inter-tool comparisons without the uncertainty that exists when two tools are compared with different datasets.

In the field of code clone detection, building labeled datasets is particularly challenging.

Unlike other domains, systems like Amazon Mechanical Turk [1] are of limited value here, because code clone labeling requires software expertise and, therefore, cannot be farmed out to large number of people. In addition, manual validation of all the possible clones would require quadratic comparisons, a combinatorial problem that becomes infeasible with growing codebases. Moreover, real world software is complex: methods can exist inside methods, or they can vary widely in size, scope, semantics and nature which, once again, demands deep expertise.

For this reason, most datasets used in code clone studies are either small or synthetically created or are labeled only for a subset of pairs. The dataset by Bellon *et al.* [18], the dataset by Murakami *et al.* [112], SOCO 2014 [41] or BigCloneBench [146] have one of the above mentioned limitations.

There has been good progress in measuring recall systematically of clone detection tools. Based on BigCloneBench dataset [146], BigCloneEval [151] estimates recall automatically by measuring how many of the labeled clone pairs are included in the output of a clone detector. However, BigCloneEval stops short of estimating precision because BigCloneBench does not contain labels for *all* possible clone pairs in it. If a clone detector identifies a clone pair that is not marked as such, only manual inspection can tell whether the pair is a false positive or a true positive. Since manual inspection is a difficult, labor intensive, and time consuming task, most clone detection approaches estimate their precision by sampling a number of their reported clone pairs, and then manually inspecting the sampled set [131, 137, 156]. So, while clone detectors report recall using BigCloneEval, the determination of their precision is still a subjective and a manual process, leading to difficulties in comparing with other tools. The lack of an established labeled dataset for precision creates a number of problems [137]:

- Precision estimation can suffer from sampling bias when sample set is small and not representative of the population.

[1]A crowd-sourcing platform to conduct tasks that demand human intelligence

- Large manual effort is required to conduct precision studies. As each pair in the sample needs to be evaluated, often by multiple judges, the total manual effort required to complete a precision study is substantial.

- The efforts put into the manual inspection of clone pairs are not reused. Each time authors want to estimate their tool's precision, they typically start from scratch.

To address these problems we present InspectorClone, an approach designed to facilitate precision studies for code clone detectors. InspectorClone helps in evaluating precision of clone detectors and, in the process, creates a dataset of well-known source code clones. InspectorClone *automatically* resolves as many clone pairs as possible, identifying a subset of pairs where manual inspection is most needed. By showing a fewer number of pairs to humans for manual inspection, InspectorClone reduces the manual effort by 40% on an average.

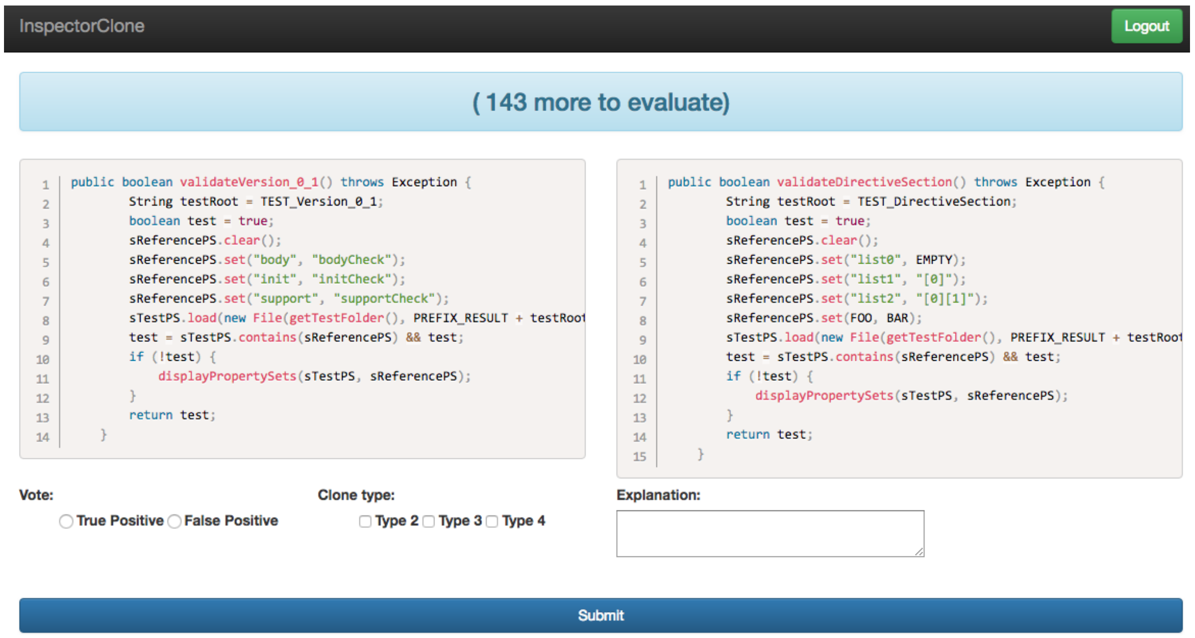
At the end, the results (automatic and manual) are aggregated to report the precision of the clone detector. Moreover, the human judgments are stored to create a manually labeled dataset of clones. This dataset is beneficial in inter-tool comparison, and also, for the exploration of machine learning and artificial intelligence techniques in clone detection. InspectorClone can be accessed at `http://www.inspectorclone.org`.

The main contributions of this work are the following:

- A semiautomated, high-precision, approach for classification of clones that reduces the manual effort of precision studies significantly.

- A publicly available web application that enables clone detection researchers to conduct precision experiments of clone detection tools and techniques.

- An evolving dataset of manually validated clone pairs that is publicly available. With



```java
public boolean validateVersion_0_1() throws Exception {
        String testRoot = TEST_Version_0_1;
        boolean test = true;
        sReferencePS.clear();
        sReferencePS.set("body", "bodyCheck");
        sReferencePS.set("init", "initCheck");
        sReferencePS.set("support", "supportCheck");
        sTestPS.load(new File(getTestFolder(), PREFIX_RESULT + testRoot
        test = sTestPS.contains(sReferencePS) && test;
        if (!test) {
            displayPropertySets(sTestPS, sReferencePS);
        }
        return test;
    }
```

```java
public boolean validateDirectiveSection() throws Exception {
        String testRoot = TEST_DirectiveSection;
        boolean test = true;
        sReferencePS.clear();
        sReferencePS.set("list0", EMPTY);
        sReferencePS.set("list1", "[0]");
        sReferencePS.set("list2", "[0][1]");
        sReferencePS.set(FOO, BAR);
        sTestPS.load(new File(getTestFolder(), PREFIX_RESULT + testRoot
        test = sTestPS.contains(sReferencePS) && test;
        if (!test) {
            displayPropertySets(sTestPS, sReferencePS);
        }
        return test;
    }
```

Figure 6.1: Validation of clone candidates on InspectorClone.

time, as InspectorClone is used, the number of humanly validated clone pairs will increase in the dataset.

This work is organized as follows. In Section 6.2, we present InspectorClone, the tool implementation of our approach. The automatic mechanisms of clone pair resolution, and related concepts are elaborated in Section 6.3, and then it is evaluated in Section 6.4. Section 6.5 presents related work, and threats to validity are explained in Section 6.6. Finally, we present the chapter summary and future work in Section 6.7.

6.2 InspectorClone

Measuring the precision of a clone detector is not a trivial process. To measure the precision, one can choose to manually validate all the clone pairs reported by a clone detector. This process, however, is extremely time consuming and impractical as the number of clone pairs reported by a tool on a standard dataset like BigCloneBench is in millions. A more practical process is to estimate the precision by humanly validating a random and statistically significant sample of clone pairs. This is what researchers do to estimate the precision of clone detectors [131, 137, 148, 156]. In this process, after running a clone detector on a dataset and getting the clone pairs, a random and statistically significant sample set of these clone pairs is assigned to multiple judges for manual inspection. The judges examine each pair to decide if it is a true clone and/or what type of clone it is. When all sampled pairs have been validated by all judges, researchers aggregate the judges' decisions, usually by taking the majority vote, and report precision.

The above process, though more practical than humanly validating every clone pair, still takes a non trivial amount of time and effort. Moreover, the effort put into one study cannot be reused in future studies. To address these issues, we present a web-based tool, named InspectorClone, that helps clone researchers in expediting the precision estimation process. InspectorClone helps by mimicking this whole process and also by automatically validating a subset of sampled clone pairs, thereby reducing the number of clone pairs shown to human judges. Moreover, by storing human judgments in a centralized database, this tool turns humans' manual effort to a long lasting resource that can be reused in future studies.

InspectorClone conducts precision studies on the dataset curated by Svajlenko et al. for facilitating recall studies [151]. Svajlenko et al. curated this dataset using IJaDataset-2.0 to conduct recall study using BigCloneEval. The dataset is available for download on InspectorClone's website. InspectorClone does not run the clone detection tool; instead, it expects users to upload the clone pairs reported by their tool (on the aforementioned dataset) to the website. The work flow is as follows.

A user, John, registers himself and his tool into InspectorClone. After registration, John can download the dataset of source code and run it on his clone detector. John, then uploads the clone pairs to InspectorClone where InspectorClone filters out the methods that are less than 50 tokens, a standard filter used in precision studies [137, 148]. John now creates an experiment to estimate the precision of his tool. He then invites multiple judges to evaluate the pairs. Once the judges are invited, InspectorClone selects a random and statistically significant sample of clone pairs. From this sample InspectorClone tries to automatically validate as many pairs as it can. All of the remaining pairs of the sample, which InspectorClone did not resolve, are then shown to the judges.

When a judge, Alice, starts an experiment assigned to her, she is shown a web page as shown in Figure 6.1. All unresolved pairs will be shown to her. This page is composed by a split screen with two columns, showing both members of a pair. The code is syntax highlighted to increase the readability. Alice must then decide if this pair does indeed represent a clone or not (if it is a true or a false positive). There are two optional form elements: one to select the clone type, and another to leave a comment. This is for the situations in which a judge is sure that a pair is indeed a clone, but has difficulty in classifying it into a type.

When all of the unresolved pairs have been validated by all judges, InspectorClone aggregates their decisions by taking the majority vote, and creates a precision report. In case there are even number of judges, InspectorClone treats a pair as a true positive only when more than 50% of the judges vote for it to be a clone pair. InspectorClone stores the human judgments in a centralized database. With time, we expect the number of humanly judged pairs to increase in this database, thereby creating a valuable asset for the community.

6.3 Automatic Classification of Clones

As explained in the previous section, to estimate the precision of a clone detection tool, InspectorClone needs to validate only a random and statistically significant sample set of the clone pairs reported by the tool. The number of these pairs in such a sample set is small and therefore, InspectorClone can use techniques which are very precise without caring much

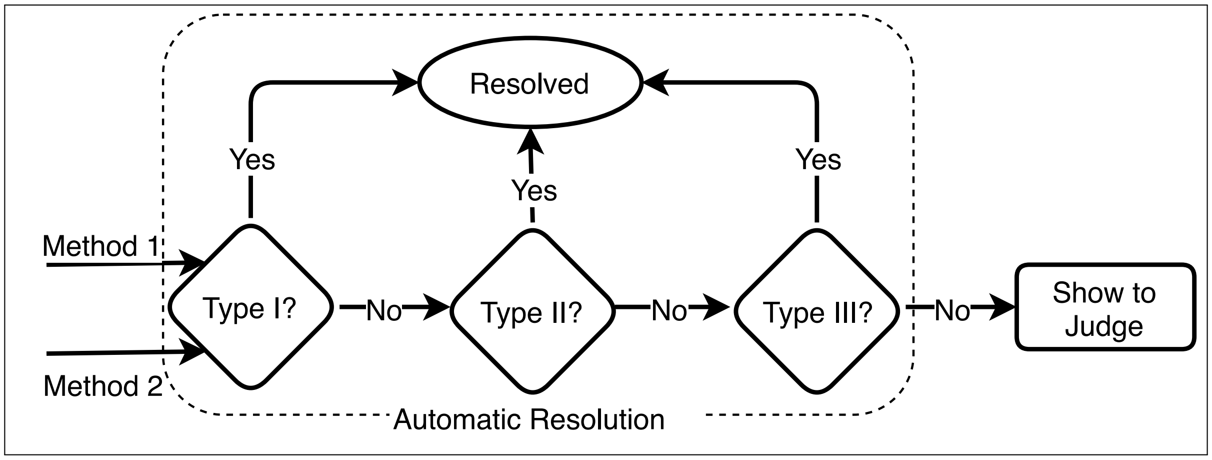


Figure 6.2: The pipeline for clone validation.

about the scalability aspects of the techniques.

Also, an automatic approach must be able to resolve pairs with very high precision; otherwise, researchers will fall back to the completely manual process. With this in mind, we designed a semiautomated approach to conduct precision studies using InspectorClone. The automatic mechanism of InspectorClone has a very high precision but it compromises on recall as it only resolves those pairs on which it has high confidence. The unresolved pairs are then shown to human judges for manual inspection. We note that clone detection tools operate at various granularities like statements, block of code, methods, files, et cetera. Also, clone detection can be carried out for software written in various languages like Java, C, C++, and Python among others. In this work, we narrow down our focus to facilitate precision studies for method level clone detectors which find clones in software systems written in Java.

6.3.1 Overview of the Approach

The methodology for clone resolution follows the pipeline presented in Figure 6.2. The two methods in a candidate pair[2] go through a series of steps in which they are checked against

[2]A candidate pair consists of two piece of code reported as clone pair by a clone detection tool. Our approach validates these pairs, and only when they are resolved as true positives they are called clone pairs.

a certain clone type. If at any step a pair is evaluated as a true clone pair, it is marked as a true positive and the system proceeds to the next candidate pair. Otherwise, if all of the steps are failed to evaluate a pair as a true positive, the pair is presented to the human judge for manual inspection. These steps are ordered by computational complexity for system performance (and also by increasing clone type complexity), and are individually described in the next sections.

6.3.2 Automatic Resolution of Type I Clones

A pair is a Type I clone if the member methods are exact replicas when neglecting source code comments and layout[3]. This makes the validation of Type I candidates similar to a simple string comparison after removing certain elements. We use Algorithm 4 to check if a candidate pair is a Type I clone. Starting with a candidate pair, the algorithm, first, removes all source code comments from both method bodies *(lines 2 and 3)*, then removes white spaces and newlines from them *(lines 4 and 5)*, and finally computes and compares the Hash (**SHA-256**), of both method bodies *(line 6)*.

Algorithm 4 Automatic Type I Resolution

INPUT: $M1$ and $M2$ are strings representing the method bodies (including method signature) of two methods for which we want to know if they are Type I clones. **OUTPUT:** *Boolean*

```
1: function IsTypeOne($M1$, $M2$)
2:     $M1$ = RemoveComments($M1$)
3:     $M2$ = RemoveComments($M2$)
4:     $M1$ = RemoveWhitespacesAndNewLines($M1$)
5:     $M2$ = RemoveWhitespacesAndNewLines($M2$)
6:     return Hash($M1$)==Hash($M2$)
7: end function
```

6.3.3 Automatic Resolution of Type II Clones

To resolve Type II pairs automatically, we use two heuristics as described below:

[3]The syntax of Java is not dependent on layout, so we can ease the definition of Type I clones. For layout-dependent syntaxes like the ones found in Python or Haskell, this approach would require a more careful deliberation

Table 6.1: Method-Level Software Metrics from [131]

Name	Description	Name	Description
XMET	# external methods called	HEFF	Halstead effort to implement
VREF	# variables referenced	HDIF	Halstead difficulty to implement
VDEC	# variables declared	EXCT	# exceptions thrown
NOS	# statements	EXCR	# exceptions referenced
NOPR	# operators	CREF	# classes referenced
NOA	# arguments	COMP	McCabes cyclomatic complexity
NEXP	# expressions	CAST	# class casts
NAND	# operands	NBLTRL∗	# Boolean literals
MDN	maximum depth of nesting	NCLTRL∗	# Character literals
LOOP	# loops (for,while)	NSLTRL∗	# String literals
LMET	# local methods called	NNLTRL∗	# Numerical literals
HVOC	Halstead vocabulary	NNULLTRL∗	# Null literals

Action heuristic: Action tokens of a method form a more stable semantic signature for the method than the identifiers or types chosen by the developer. This is because identifiers and types often change in duplicating methods, while Action tokens tend to remain the same. The reason is that methods and class attributes, represented by Action tokens, bring pre-implemented functionalities, which reduce the burden of coding, and hence, are not probable to be removed or modified after cloning.

Metric heuristic: Software metrics, measuring different characteristics of source code, can capture structural information of a method. These measurements are resilient to changes in identifier names and literals – a useful property in the detection of Type II clones. Hence, we use 24 method level software metrics shown in Table 6.1 for Type II resolution. The details of these metrics can be found elsewhere [131, 134]. A detailed explanation about the application of Action tokens and software metrics in clone detection can be found in [131] and also in Section 4.2.

We use Algorithm 5 to check if a candidate pair is a Type II clone. First,we get a list of action tokens for both methods *(line 2 and 3)*. Then, we compare if these lists are identical *(line 4)*, that is, the contents along with their order of appearance in these lists match. If the lists are identical, we get a list of metrics for both methods *(line 5 and 6)* and then return true if these lists are identical *(line 7)*, else return false.

This algorithm ensures that a candidate pair is resolved as Type II only when there is a 100% match in both the metrics and the Action tokens. The rational is that Type II clones differ in identifier names and literal values while their structure (captured by metrics), and their method calls and accessed class fields (captured using Action tokens) remain the same.

Algorithm 5 Automatic Type II Resolution

INPUT: $M1$ and $M2$ are strings representing the method bodies (including method signature) of two methods for which we want to know if they are Type II clones. **OUTPUT:** *Boolean*

```
1: function IsTypeTwo($M1$, $M2$)
2:     $ListATofM1$ = GetActionTokens($M1$)
3:     $ListATofM2$ = GetActionTokens($M2$)
4:     if IsIdentical($ListATofM1$,$ListATofM2$) then
5:         $ListMetM1$ = GetMetrics($M1$)
6:         $ListMetM2$ = GetMetrics($M2$)
7:         return IsIdentical($ListMetM1$,$ListMetM2$)
8:     end if
9:     return $False$
10: end function
```

6.3.4 Automatic Resolution of Type III Clones

Typically, syntactic clone detectors detect clones in ST3 and VST3 categories. This is because there is a high probability that code snippets with less than 70% syntactical similarity are coincidentally similar. Moreover, to detect clones between 0-70% similarity range, detectors may need to capture the semantic similarity, a harder problem in clone detection. Therefore, achieving very high precision in the automatic resolution of all of these subcategories is considered hard. As we want to make sure whatever candidate pair our approach resolves as a clone pair, is indeed a true clone pair, we focus on the first two subcategories, namely VST3 and ST3.

To resolve Type III candidates automatically, we first make the candidate pairs go through an *Action filter*, and the ones that survive this filter are fed to a deep learning classifier that predicts whether they are true clone pairs. Placing *Action filter* before the classifier ensures that candidate pairs, whose methods do not share a specific amount of action tokens, are filtered out early, and shown to judges instead of being resolved automatically; hence, increasing precision.

In the following subsections we first explain the training set used in training the deep learning model, and next, we describe the details of the trained model. Finally, we provide the results of a sensitivity analysis we did for selecting the proper *Action filter* threshold to resolve Type III clones with high precision.

6.3.4.1 Dataset Curation

Since the machine learning classifier is supposed to resolve Type III candidates, we need a training set with clone pairs from this category. Also, to resolve clone pairs with very high precision, we want our dataset to contain true clone pairs which are very similar in terms of both their semantics and structure. To generate such dataset, we use two token based state of the art clone detectors, CloneWorks (Aggressive mode) [153] and SourcererCC [137]. The configurations of these tools are shown in Table 6.6. These clone detectors detect Type III clone pairs up to ST3 category, where the methods in each clone pair have high structural similarity. On the other hand, we ensure high semantic similarity in the methods of each clone pair in our dataset by using Action Filter with threshold set to 90%. The starting dataset used to generate our training dataset is BigCloneBench.

The numbers related to dataset creation process are reported in Table 6.2. The training set includes equal number of both assumed positives (clones) and assumed negatives (non-clones). To get the set of assumed positives, we took an intersection of the clone pairs detected by the two tools (RowId 3 in Table 6.2). We then removed all Type I and Type II pairs from this intersection and selected the pairs which satisfy our Action filter (RowId 4). To ensure that these pairs are true clone pairs, we randomly sampled 1,851 pairs (a

Listing 6.1: Example VST3 Clone Pair

```
 1 private boolean doPurgeStudy(DirWriter w, DirRecord parent, int[] counter) throws IOException {
 2     boolean matchAll = true;
 3     LinkedList toRemove = new LinkedList();
 4     for (DirRecord rec = parent.getFirstChild(true); rec != null; rec = rec.getNextSibling(true)) {
 5         if (doPurgeSeries(w, rec, counter)) {
 6             toRemove.add(rec);
 7         } else {
 8         matchAll = false;
 9         }
10     }
11     if (matchAll) {
12         return true;
13     }
14     for (Iterator it = toRemove.iterator(); it.hasNext(); ) {
15         counter[0] += w.remove((DirRecord) it.next());
16     }
17     return false;
18 }
19 ----------------------------------
20 private boolean doPurgeInstances(DirWriter w, DirRecord parent, int[] counter) throws IOException {
21     boolean matchAll = true;
22     LinkedList toRemove = new LinkedList();
23     for (DirRecord rec = parent.getFirstChild(true); rec != null; rec = rec.getNextSibling(true)) {
24         File file = w.getRefFile(rec.getRefFileIDs());
25         if (!file.exists()) {
26             toRemove.add(rec);
27         } else {
28         matchAll = false;
29         }
30     }
31     if (matchAll) {
32         return true;
33     }
34     for (Iterator it = toRemove.iterator(); it.hasNext(); ) {
35         counter[0] += w.remove((DirRecord) it.next());
36     }
37     return false;
38 }
```

statistically significant sample with 99% confidence level and 3% confidence interval), and validated them manually. Two judges, who are also the authors of this paper, independently went through these clone pairs and unanimously found all pairs to be true clone pairs.

Listing 6.1 shows an example of true clone pair (VST3) found by the judges. Both methods in this example seem to have a very similar aim: first, they fill a *LinkedList (line 6 and 26)* and then they iterate over the *LinkedList* to remove its contents *(lines 14 to 16 and lines 34 to 36)*. The methods not only share many Action tokens, their structures also look very similar. Moreover, the line and token similarity between the methods are high, making this pair a good example of a true positive.

Table 6.2: Dataset Creation Process Statistics

RowId	Dataset	Number of Pairs
1	SourcererCC Pairs	909,409
2	CloneWorks Pairs	8,053,303
3	SourcererCC & CloneWorks Intersection	699,389
4	Intersection after Removal (Clone Pairs)	**53,058**
5	Non-clone pairs at 90% *Action filter*	18,195,489
6	Union of Pairs by SourcererCC, CloneWorks, Nicad	8,408,734
7	Non-clones after Removing Union Pairs	18,135,188
8	Non-clones after Random Sampling	**53,058**
9	Total Rows in Final Dataset	**106,116**

The training set needs not only positive samples of clones, but also negative ones. Getting these pairs is considerably more difficult– while there is an enormous amount of code pairs that are not clones of each other, for machine learning purposes, it is not useful to include pairs that have no similarities whatsoever. Ideally, we would like to include pairs that we know with high certainty are not clones, but that are sufficiently similar that they could be confused as clones.

To get such assumed negative pairs, we modified Oreo [131], a clone detector designed to detect Type III clones even in harder clone categories, to predict non-clones such that they have at least 90% similarity in their Action tokens (RowId 5). The original source code of Oreo is available at [130]. Then, we took a union of the clone pairs reported by three state of the art clone detectors: CloneWorks, SourcererCC, and NiCad [127] (RowId 6). NiCad's configurations are shown in Table 6.6. To ensure high confidence in the non-clone pairs, we removed any non-clone pair which is present in the union set (RowId 7). Finally, we did a manual analysis similar to what we did for true positives to gain more assurance about the non-clone pairs. The same two judges, independently as before, went through a random sample of 400 pairs. They found many examples which were definitely non-clones, and also found some examples of MT3 and WT3/4 clones, where the pairs shared high semantic similarity but the structural similarity was weak. This is useful in increasing the precision

Listing 6.2: Example MT3 Clone Pair

```java
public static void copy(InputStream i, int buf, OutputStream o) throws IOException {
    byte b[] = new byte[buf];
    for (; ; ) {
        int g = i.read(b);
        if (g == -1) break;
        o.write(b, 0, g);
    }
}
--------------------------------
public static void copyTo(InputStream in, OutputStream out) throws IOException {
    byte buffer [] = new byte[2048];
    int n = 0;
    while ((n = in.read(buffer)) != -1) {
        out.write(buffer, 0, n);
    }
    out.flush();
}
```

of the machine learning model since it learns to classify these harder pairs, which are closer to the threshold boundary, as non clones. This is a desirable behavior as these harder cases are then left for human judgment.

Listing 6.2 shows an example of an MT3 pair found by the judges. Both methods in this pair are semantically similar as they both intend to copy the contents from an *InputStream* to an *OutputStream*. The structural and token similarity between the two methods, however, is low, making it harder to detect as a clone pair by many token based clone detectors. Similarly, Listing 6.3 shows another pair that semantically, are performing the same task, but the structural similarity between the two methods is very low. Such methods are good candidates that should be left for human judgment.

We then took a random sample of 53,058 pairs (RowId 8) from the above obtained pairs (RowId 7) to have the number of non-clone pairs matched with the number of true clone pairs (RowId 4). Finally, we aggregated the pairs from RowId 4 and RowId 8 to create a dataset (RowId 9) which we used for training and validating the machine learning model. Each row of this finalized dataset contains a method pair represented as a vector of 48 metrics (24 metrics of Table 6.1 for each method), and a label denoting whether this pair is a clone or not.

Listing 6.3: Example WT3/4 Clone Pair

```
protected void copy(InputStream _in, OutputStream _out) throws IOException {
    byte[] buf = new byte[1024];
    int len = 0;
    while ((len = _in.read(buf)) > 0) _out.write(buf, 0, len);
}
---------------------------------
public static long copy(InputStream is, OutputStream os) throws IOException {
    byte buffer [] = new byte[1024];
    int readed;
    long length = 0;
    while ((readed = is.read(buffer)) != -1) {
        if (readed > 0) {
            length += readed;
            os.write(buffer, 0, readed);
        } else {
        TimerHelper.notSafeSleep(100);
    }
}
return length;
}
```

6.3.4.2 Deep Learning Model

To classify Type III clones we are using Siamese architecture to train a Deep Neural Networks (DNN) model. We carried out model comparison analysis on the training dataset at hand and found Siamese model to outperform other models. Model comparison results are explained later in this section. The architecture of this model is same as the model we trained for Oreo. The differences between the two models are mostly driven by the purpose for which we want to train these models. For Oreo, we wanted a good balance between recall and precision but in InspectorClone our goal is to maximize the precision. To fulfill this need, we curated a separate train dataset for InspectorClone's model. The dataset is smaller in comparison to Oreo's training dataset and therefore the number of neurons in each component of InspectorClone is relatively smaller than that of Oreo.

Figure 6.3 shows the architecture of the Siamese model we trained for our approach. It consists of three components: i) two identical subnetworks, ii) a comparator unit, and iii) a classification unit. The input to the model are the feature vectors of each method (24 metrics) in the candidate pair. These feature vectors are then transformed and processed by the two subnetworks and the comparator. Finally, the classification unit outputs a number

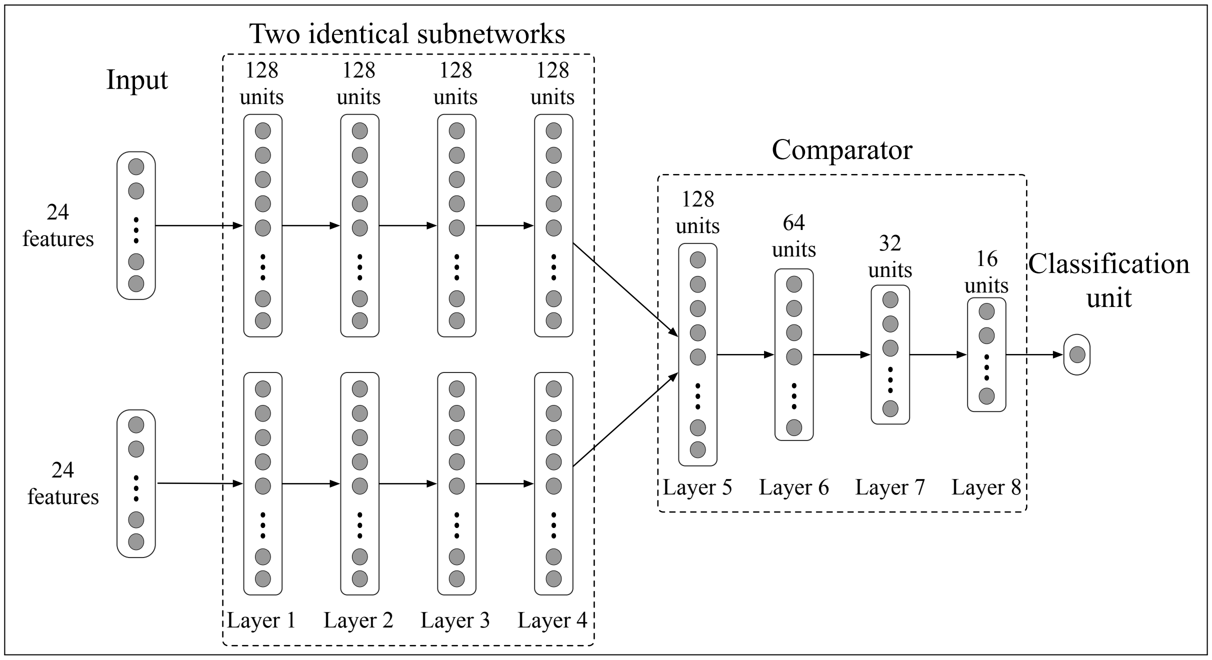


Figure 6.3: Siamese architecture

Table 6.3: Precision and Recall on the test set for the Siamese Neural Network model using different thresholds.

Threshold	Precision	Recall
0.6	0.970	0.920
0.7	0.981	0.882
0.8	0.989	0.844
0.9	0.996	0.700

between 0 and 1, representing the probability of a pair being a clone. A more detailed explanation of this architecture and its components can be found elsewhere [131].

To perform the model selection experiments, the training dataset of 116,000 code pairs is randomly divided into 80% for training and 20% for testing. Furthermore, 5,000 pairs from the training set are set aside for validation purposes, i.e. for hyperparameter tuning. First, we explored the hyper-parameter tuning for the Siamese model. The best performing model is the one shown in Figure 6.3. Each of the two subnetwork layers in this model has 4 layers, each with 128 neurons and the comparator has four fully connected layers of sizes 128-64-32-16. The output of the comparator is then fed into a single classification neuron

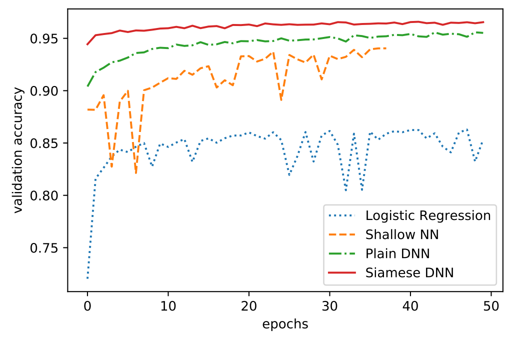


Figure 6.4: Validation Accuracy

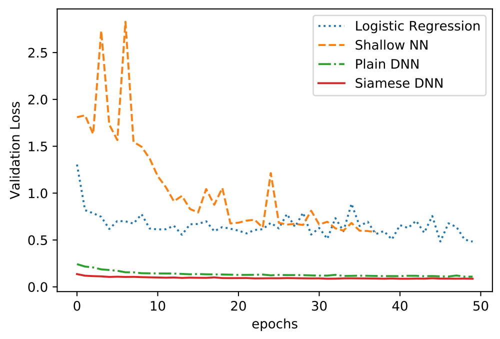


Figure 6.5: Validation Loss

with sigmoidal (logistic) activation function. The output of this neuron is a value between 0 and 1. Normally, the values more than 0.5 are assigned label 1, and values less than 0.5, are assigned label 0. However, since our goal in this problem is to achieve almost perfect precision (so that the number of false positives tends to zero), during production (testing), we set the threshold to announce a pair a true clone pair to be 0.9. The code pairs which have prediction values between 0 to 0.9 are sent to human judges for further inspection. Table 6.3 shows the Precision and Recall values on the test set at different thresholds. As it is observed, using this deep learning approach with a threshold of 0.9 yields almost perfect precision (0.996).

We also compared the Siamese DNN to other models with different architectures, including: (1) a plain fully connected neural network (Plain DNN) with similar number and sizes of hidden layers as the Siamese one; (2) a shallow neural network (Shallow NN) with one hidden layer and similar number of parameters; and (3) a logistic regression model. Since our final goal is to achieve a high Precision, all the comparisons are done when the thresholds for all models are set to be 0.9. We first compared their performance during the training stage. Figure 6.4 shows that the Siamese DNN outperforms the other models in terms of accuracy on the validation set. The accuracy of the Siamese DNN converges to 96.0%, while the accuracy for plain DNN and Shallow NN converges only to 94.0%. Figure 6.5 shows that for the validation loss also, the Siamese structure is superior to the other models. The average

Table 6.4: Precision and Recall values on test set

Threshold=0.9	Precision	Recall
Logistic Regression	0.984	0.828
Shallow NN	0.991	0.734
Plain DNN	0.983	0.841
Siamese DNN	0.996	0.700

loss value for the Siamese DNN converges to 0.103 (as apposed to Plain DNN: 0.139; Shallow NN: 0.162; Logistic regression model: 0.222). Thus in short, the Siamese DNN better fits the training data.

Next, the model performance is compared at the testing stage. Table 6.4 shows that the Siamese network has the highest Precision, equal to 0.996. In short, this shows that the Siamese network has better generalization performance than the other models used in the comparison.

6.3.4.3 Sensitivity Analysis

We did a sensitivity analysis to find the optimum threshold of *Action filter* with the goal of not having any false positives, and maximizing the number of pairs resolved automatically.

Methodology. We used the clone pairs reported by SourcererCC on the BigCloneBench Dataset. We ran InspectorClone with four different threshold values of *Action filter*: 60%, 65%, 70%, and 75%. One author manually inspected the clone pairs that are automatically resolved by InspectorClone to figure out the number of false positives in them. Results of this analysis are denoted in Table 6.5. The first column of this table shows the examined thresholds, and the next three columns, respectively, denote the number of automatically resolved Type I, Type II, and Type III clone pairs. The next three columns show the number of false positives observed at each clone category, and the last column depicts the number of clone pairs that need the manual validation by humans. At 60% and 65% thresholds, we

Table 6.5: Sensitivity Analysis Statistics

Threshold	Auto Type III	Auto Type III FP	Manual
60%	124	16	68
65%	78	17	121
70%	49	0	149
75%	18	0	174

observed some false positives, whereas at 70% and 75% thresholds, no false positives were observed. The number of automatically resolved clone pairs at 70% threshold (49) is greater than this number at 75% threshold (18). Consequently, this threshold was selected to be used in *Action filter*.

6.4 Evaluation

As discussed earlier, the main goal of our approach is to automatically resolve as many clone pairs as possible with high precision. Hence, to evaluate it, we designed an experiment using InspectorClone, and seven clone detectors. The goal of this experiment is twofold: i) to understand the impact of our approach on the reduction of manual effort, and ii) to measure the precision of the automatic clone resolution approach.

We include SourcererCC, iClones [47], NiCad, CloneWorks, SimCad [155] as popular examples of modern clone detectors that support Type III clone detection. CloneWorks comes in two different modes, Aggressive and Conservative; we tested InspectorClone on both of these modes. We also include two recent tools, Oreo and CCAligner [156]. While all these tools detect clones in Type I, Type II, and early categories of Type III (*VST3* and *ST3*), Oreo and CCAligner are capable of detecting clones beyond *ST3* categories such as *MT3*. We also wanted to include Deckard [64] and CPD [1]; however, they both detect clones beyond method boundaries. At this time, we cannot reliably conduct a meaningful experiment with them on InspectorClone, which only supports method level clone detectors as of now. Also, both of these tools report their results as clone classes and not as clone pairs. When we ran processes to generate clone pairs from these clone classes, they both produced large amount of clone pairs. We killed the processes after generating more than 175G of clone pairs for each of them as these are very big files for InspectorClone to process.

We ran all tools on the recall dataset of BigCloneBench and obtained the clone pairs reported by each tool. We then uploaded the clones reported by each tool to InspectorClone, and calculated the number of clone pairs automatically resolved in each category, and the number of pairs left for manual validation. We configured InspectorClone to consider only those pairs that have methods with at least 50 language tokens, a standard size filter used in precision studies [131, 137].

To gain high confidence in our experiment results, we conducted 2 rounds of experiments for each tool (a total of 16 experiments, with 7 tools and CloneWorks being executed in two modes). In each round, InspectorClone sampled 400 random candidate pairs from the output of each tool. InspectorClone then automatically resolved some clone pairs as assumed positives, leaving the rest for manual validation. To measure the precision of the automatic resolution part, five judges, who are also authors of this paper, independently went through the whole set of 2,545 automatically resolved clone pairs to look for possible false positives. The judges were also asked to report the time they took to complete each round of experiment. In total, it took around 58 man hours to complete all 80 experiments (16 rounds per each judge).

The results of this experiment are shown in Table 6.6. The first column shows the name of the tool. The next three columns denote, respectively, the number of Type I, Type II, and Type III candidate pairs that were automatically resolved by InspectorClone. The fifth column shows the number of candidate pairs that could not be automatically classified, and needed manual validation by humans (out of the sample of 400). The sixth column

Table 6.6: Reduction of Manual Effort

Tool	Automatically Resolved			Manual Inspection	FP	Tool Configuration
	T1	T2	T3	(out of 400)		
CCAligner	18	38	60	284	0	MIL=6, $\Theta = 60\%$
	18	31	64	287	0	e=1, q=6
CloneWorks(A)	118	27	34	221	0	MIT=1, $\Theta = 70\%$,
	118	35	37	210	0	Mode=Aggressive
CloneWorks(C)	53	43	36	268	0	MIT=1, $\Theta = 70\%$,
	54	58	26	262	0	Mode=Conservative
iClones	254	59	26	61	0	MIT=50,
	256	63	15	66	0	min block=20
NiCad	99	35	159	107	1	MIL=6, BIN=True,
	115	26	165	94	0	IA=True, $\Theta = 30\%$
Oreo	0	0	0	400	0	MIT=15,$\Theta = 55\%$,
	0	0	0	400	0	$\Gamma = 60\%$
SourcererCC	155	25	8	212	0	MIT=1,
	149	24	12	215	0	$\Theta = 70\%$
SimCad	15	0	2	383	0	GT=True,
	10	2	3	385	0	US=True,MIL=6

(*FP*) contains the number of false positives (after considering majority vote) observed by human judges in the automatically resolved pairs. And finally, the seventh column shows the configurations which were used to run the tools. These configurations are based on our discussions with their developers, and also the configurations suggested in [148]. In the table, *MIT* stands for minimum tokens, *MIL* stands for minimum number of lines, *BIN* and *IA*, respectively stand for blind identifier normalization and literal abstraction used in NiCad. Θ stands for similarity threshold (for NiCad, it is *difference threshold*, and for Oreo it is *Action filter* threshold), Γ is the threshold for input partition used in Oreo. In CCAligner's configurations, e stands for *edit distance*, and q is the *window size*. *GT* and and *US* stand for *greedy transformation* and *unicode support* used in SimCad.

As the table shows, InspectorClone reduced the number of pairs that need manual analysis

Listing 6.4: Example Candidate Pair from Oreo

```
public static <T> T readStreamAsObject(InputStream inputStream, Class<T> type) throws
        ClassNotFoundException, IOException {
        ObjectInputStream objectInputStream = null;
        try {
            objectInputStream = new ObjectInputStream(inputStream);
            return type.cast(objectInputStream.readObject());
         } finally {
        Utility.close(objectInputStream);
    }
}
--------------------------------
public static <T extends Serializable> T deserialise(Class<T> class1, File out) throws
        ClassNotFoundException {
        try {
            FileInputStream fis = new FileInputStream(out);
            ObjectInputStream in = new ObjectInputStream(fis);
            Object output = in.readObject();
            in.close();
            return class1.cast(output);
         } catch (IOException ex) {
        ex.printStackTrace();
        return null;
    }
}
```

for all tools except for Oreo. On an average, there is a 39% reduction in the number of clone pairs that are left for human judges. Most reduction is observed for iClones (84%) and NiCad (74%), while for SimCad (4%) and Oreo (0%), we observed little to no reductions. The reduction for rest of the tools ranges from 28% to 47%.

To understand why InspectorClone did not help in reducing the number of pairs for Oreo and SimCad, two judges went through the samples of one of the two experiments conducted for both tools. For SimCad, the judges reported 358 out of 400 pairs as false positives of the SimCad tool itself (10.5% precision). The presence of large number of false positives in the pairs of SimCad explains why InspectorClone did not help much in resolving its pairs. For Oreo, the judges reported a much higher precision of 80%, where they reported 80 out of 400 pairs as false positives of the tool. Almost all of the clone pairs in the sample of Oreo were found to be in harder to detect Type III categories (*MT3* and *WT3*). An example of such a pair is shown in Listing 6.4. Both methods in this example are reading an *object* from an input stream, and then, they cast this object into the *type* they received in their arguments. Though they are performing similar tasks and hence, are semantically similar, they differ significantly in their structural properties. This qualifies such pairs to fall in harder to detect MT3/WT3 categories, making them good candidates for human inspection.

If we remove Oreo and SimCad, which are two special cases, from the analysis, on an average, InspectorClone resolves 52% of the clone pairs. The results demonstrate that InspectorClone can have a key role in reducing the burden of manual effort needed by users in precision studies.

Apart from the reduction in manual effort, the precision of the automatic classification is of a great importance. Out of 1,432 Type I, and 466 Type II clone pairs resolved by InspectorClone, judges found no false positives, giving InspectorClone perfect precision scores in these categories. The judges reported some false positives in the Type III pairs. We report the precision for InspectorClone with following two strategies: i) Strategy-A, when majority vote is considered (numbers in column 6 of Table 6.6 are based on this strategy), and ii) Strategy-B, when a pair is considered false positive if any of the judges report it as a false positive. In Strategy-A, one false positive was found out of 647 Type III pairs, giving InspectorClone a precision score of 99.8%. With this strategy, the precision of InspectorClone for all types of pairs combined (2,545 pairs) is 99.96%. The methods in this false positive pair are big in size (≈140 *NOS*). Both of these methods make around 100 calls to *add()* method of a list object, which results into a high match in their Action tokens. Also, the arguments to these *add()* method calls in both of these methods are *String Literals*, thereby increasing the match count in the *NSLTRL* metric, which in turn contributes to a high structural match, making InspectorClone resolve the pair as a Type III clone. However, the *String Literals* are very different and there exists a *loop* in one of the methods, which led the judges to mark this pair as a false positive.

In Strategy-B, 11 false positives were found, giving InspectorClone a precision score of 98.3% in Type III pairs. If pairs of all types are combined, this strategy gives a precision score of 99.57%. In their judgments of 2,545 pairs, the judges were unanimously in agreement on

2,534 pairs.

When asked about these false positives, all judges mentioned that except for two or three pairs, all of these pairs are borderline cases. For instance, one judge noted: *"I am on the fence about this pair"*. And for a different pair another judge noted: *"I hesitate if it is a clone or not"*. This shows that identification of clones is a subjective task which involves cases that are hard to judge even by humans. We note, that the judges are well aware of the clone definition and clone types and all of them have previously contributed to the research involving software clones or clone detectors.

The results show that the strict thresholds used for automatic clone validation are appropriate, if not prefect, and that we can rely on the automatically resolved pairs with high confidence.

6.5 Related Work

Measuring the detection capabilities of clone detection tools is an important part of source code cloning research. This demands the existence of labeled and standardized datasets that can assist with this measurement. Therefore, development of such datasets have been the focus of research throughout the years. Unlike the vast majority of areas for which the tasks for producing labeled datasets are accessible to a large number of people (without any special expertise being required), developing labeled datasets related to source code cloning requires significant expertise in a narrow topic: programming. For example, image or speech recognition can be done by everyone; some examples of platforms that assist people with these tasks are Amazon Mechanical Turk, or the population of College students. However, such platforms cannot be used in preparation of source code cloning datasets due to the need for the related knowledge. For this reason, researchers have tried to build such labeled datasets in other ways. Most of these works have been successful in estimating the recall since recall estimation does not require the comprehensive labeling of all pairs, which is needed in measuring the precision. Here, we briefly discuss a set of these efforts.

BigCloneBench (BCB) dataset [146, 148] is probably the most related work to ours. We have also used it in the evaluation of our approach with InspectorClone. The underlying corpus of Java source code used by BCB is IJaDataset-2.0[4]. This dataset represents a large inter-project Java repository containing 25,000 open source projects, with 2.3 million source files and 365M lines of code [148]. BCB contains a subset of IJaDataset curated by human judgment, and it contains over 8 million known clone pairs within IJaDataset. It is the result of using IJaDataset, selecting a series of known algorithms (sorting algorithms is one example), and tracking possible implementations of these across the dataset. Hence, not all possible clone pairs are tagged in this dataset, and there exist many pairs that are not tagged. As a result, this dataset cannot be used to measure the precision of clone detectors, but it has been used by BigCloneEval (BCE) [151] to estimate the recall of clone detector tools automatically.

Another dataset is created by Bellon *et al.* [18]. To prepare this dataset, Bellon manually validated 2% of the clones reported by then (year 2002) contemporary clone detectors for eight software systems. Svajlenko *et al.* [148] found that this benchmark is not suitable for accurate evaluation of modern clone detection tools. They attributed many of the problems in the dataset to it being built using tools that are now outdated. It has also been found to have other problems as we see next.

Murakami *et al.*'s dataset [112] is an improvement on the Bellon *et al.*'s dataset. Murakami *et al.* found out that since Bellon dataset does not contain locational information of gapped lines (i.e. lines that are present in a pair but missing in the other), it has not evaluated some Type III clones correctly. Hence, they added this information and improved the dataset with

[4]Available at `https://sites.google.com/site/asegsecold//projects/seclone` (March 2018).

this respect.

Another effort has been made in SOCO 2014 [41]. SOCO was a challenge defined for detection of source code pairs that are reused. The task was carried out at document level, and in C/C++ and Java. Two datasets were provided: train and test. Train dataset was labeled and used to train an algorithm that can find source code pairs in which one pair is developed reusing the other one. The test dataset was used to evaluate the accuracy of the developed algorithm with respect to recall, precision, and F1[5]. SOCO contains only 259 Java files and 79 C files, and these examples do not represent realistic software projects (the origin of the source code is unclear).

6.6 Threats to Validity and Limitations

The measurement accuracy of our approach, and its reduction in manual effort was performed manually and independently by five expert judges over a large sample of clone pairs detected by seven different clone detection tools. However, these five judges were also authors of this work and more importantly, like any work that relies on human action, practical limitations related to bias and cognition could have affected our analysis. We mitigated this issue by strictly adhering to the definition of the clone types during manual classification and also by sharing the data for researchers to verify.

The tools used to generate clone pairs and validate our approach can have an impact on the validation of our approach. For example, if a tool has a tendency to detect large clones, then the validation will be performed on the large clones too. To compensate for this bias and to gain more confident in our approach we evaluated it with seven different clone detectors.

Another important consideration is that our approach focuses on Java methods and is eval-

[5]The F1 score is a measurement provided by the harmonic mean between precision and recall.

uated for methods with 50 tokens or more. It is possible to apply this methodology to other granularities of source code and to methods smaller than 50 tokens, but doing so would require modifying the existing components of our approach, specifically the software metrics.

We measure manual effort involved in precision studies as the number of clone pairs that need manual inspection. The effort, however, to inspect clone pairs of different types and sizes varies significantly and therefore may not be linear to the number of pairs.

6.7 Chapter Summary

We have presented a semiautomated approach and a tool, InspectorClone, that facilitate precision studies. We evaluated the precision of the automatic clone resolution part of this approach on seven different clone detectors. Our experiments show that the precision of InspectorClone is very high (¿99.5%) making it suitable for conducting precision studies. Further, we demonstrated that the number of clone pairs resolved by InspectorClone is significant.

InspectorClone is available to the community and it provides a beneficial framework to access community efforts and to contribute back to them.

As future work, we are looking at the implementation of this approach in different programming languages, with different granularities (classes or files instead of methods for example) and on different scales (clone with less than 50 tokens).

Chapter 7

Conclusion

7.1 Dissertation Summary

There has been a tremendous increase in the amount of source code available today in the Web-hosted open source repository services such as GitHub, BitBucket and SourceForge. Such large corpora present the opportunity to improve the state of the art in various modern use cases of clone detection. Some of these use cases include studying the extent of cloning within and across code hosting repository platforms [99, 162], studying patterns of clone evaluation [17, 79], conducting empirical studies to understand the characteristics of clones [133], detecting similar mobile applications [28], license violation detection [44, 85], mining library candidates [60], reverse engineering product lines [44, 56], code search [74], and finding the provenance of a component [32]. Though these opportunities are exciting, scaling such vast corpora poses critical challenge making scalability a must-have requirement for modern clone detection tools. SourcererCC and CloneWorks are two such tools to have addressed the scalability related challenges. These tools are designed to detect clones upto early Type III subcategories, namely VST3 and ST3. However,in our experience, we have observed that there exist more clone pairs in the Twilight Zone (MT3 and WT3 subcategories) than in the early Type III subcategories, making it desirable to push the boundaries of scalable clone detection to cover the Twilight Zone. Maintaining high accuracy while addressing the prohibitive problem of candidate explosion makes scalable clone detection in the Twilight Zone a hard problem. This dissertation presents Oreo, a tool to conduct accurate and large scale clone detection in the Twilight Zone. Oreo uses concepts from software metrics, information retrieval and machine learning to address the scalability challenges while maintaining high recall and precision across Type I, Type II, Type III as well as in the Twilight Zone.

To address the challenge of candidate explosion I proposed a novel semantic filter named Action filter along with two level input partitioning to filter out a large number of unlikely clone pairs. In an experiment to demonstrate scalability, we found Oreo scales to IJaDataset, a large inter-project repository containing 25,000 open-source Java projects, and 250 MLOC on a standard workstation. Oreo is the only clone detector designed to operate in the Twilight Zone which has shown to scale this dataset.

We measure Oreo's recall using BigCloneBench, a standard benchmark of real world software clones. Oreo performs at par with the state of the art near miss Type III clone detectors on Type I, Type II, and early categories of Type III clones. Oreo performs significantly better than the other accurate Type III clone detectors by finding an order of magnitude more clones in the Twilight Zone. In the blind experiments consisting of two judges, we found that Oreo's precision is at par with the other state of the art tools in the Type I and Type II clone categories. Oreo's precision is found to be 82.5%, which is lower than the best performing clone detectors in the Type III clones. Given that Oreo detects clones in the Twilight Zone, a territory where other tools are not designed to operate, a precision of 82.5% is considered to be very good. The only other tool designed to find clones beyond the ST3 category is CCAligner, with both recall and precision (71.25%) significantly lower than that of Oreo.

7.2 Lessons Learned

During the development of Oreo I experimented with different machine learning algorithms before settling with the Deep Neural Networks. We ran our experiments with multiple classifiers: K-Nearest Neighbors (KNN) [160], Naive Bayes [160] , Classification and Regression Trees (CART) [160], Logistic Regression (LR) [36], Linear Discriminant Analysis (LDA) [62]. To this end, we used the dataset of 3,562 Java projects which I created in one of my previous work [133]. The dataset consists of 3,562 Java projects hosted on Maven [104]. The comprehensive list of projects with their version information is available at `http://mondego.ics.uci.edu/projects/clone-metrics/`.

During my experiments with these algorithms, I learned several important lessons which I summarize below.

- Software metrics have a great potential for being used in finding code clone pairs. It is important to manually inspect the true positives, false positives, true negatives and false negatives produced by such approaches to get insights into how to engineer the features using software metrics.
- For many algorithms in machine learning it is advised to remove the features which are highly correlated [53]. Random-Forrest however, does not get affected much by correlated features. The decision-trees used in Random-forest, during training, calculate the information gain for each feature at every step and then learn to make a decision. The highly correlated features, at a given step, will have a very similar information gain and therefore only one of them will be used to make a decision on that step. This makes having correlated features redundant yet not affecting the model negatively. As a result,

we did not have to analyze the correlations among features heavily. However, we found some interesting patterns in features that needed attention: We noticed that NLOC and NOS values for a very similar code could be different. This is because NLOC captures the number of lines of code, whereas NOS captures the number of statements in code. For example, one could write three statements in one line, decreasing the NLOC value without affecting the number of statements, which will still be three. Thus, NLOC would not have useful information gain for our model; hence, we removed NLOC and reserved NOS. We also removed MOD, which captures the number of modifiers in a method. It is a binary feature (0 or 1) and we noticed that it has no correlation with *is_clone* label.

- While analyzing false negatives we noticed that *Percentage Difference* between the metric values is a good feature but there were many false negatives in our predictions. Analyses showed that percentage difference could mislead our model while training; let's see this with the help of an example. Consider a candidate pair M_1, M_2 where M_1 has n_1 loops and M_2 has n_2 number of loops such that $n_1 > n_2$. The percentage difference between their number of loops, is given by following expression.

$$\frac{n_1 - n_2}{n_1} * 100 \tag{7.1}$$

The percentage difference does not scale well when we increase n_1 and n_2. If n_1 is 2 and n_1 is 1, the percentage difference is 50. However, if n_1 is 10 and n_2 is 9 the percentage difference is 10. Notice in both cases the absolute difference is 1. Also, if n_2 is 0, then for any natural number value of n_1, the percentage difference will be 100%; so there is a definite need of feature scaling. We addressed this by using a very simple technique: We added a constant c to both n_1 and n_2, where c is a natural number. For most of our metrics which needed scaling, their metric values were in range of 0 to 15. Hence, a very big value of c would have removed important information from the features and a small value like 1 would not scale the features. Considering this and assessing different numbers, we decided to set c to 11. The result of this was to use the formula shown in 7.2 for Feature Engineering.

$$Percentage - diff(f1, f2) = \frac{|f1 - f2|}{Max(f1, f2) + 11} * 100 \tag{7.2}$$

- Random Forest algorithm performs better than other non DNN algorithms for this problem on the dataset and metrics that we used.
- Analysis of false positives revealed many code fragments that are not real clone pairs, but have identical structures. Hence, one needs additional features that beside capturing the structural similarity, can recognize semantic similarity.
- Machine learning approaches are susceptible to the problem of *Candidate Explosion* as every possible method pair needs to be compared in order to classify pairs as clones or non clones.
- Action filter not only addresses the candidate explosion problem but also ensures that the method pairs that pass this filter share high semantic match. This removes/mitigates the need of capturing semantic features making feature engineering easier and faster.
- Concepts from information retrieval such as creating in-memory indexes can speed up the detection process by many folds. In Oreo I use two level input partitioning where I create self contained partitions such that I can efficiently create in-memory indexes and restrict the query to only one partition. This strategy not only helped me in reducing the number of candidates, but also in fast retrieval of candidate clones.

7.3 Future Work in Clone Detection

This dissertation emphasized on the importance of developing scalable and accurate code clone detectors. To this end we presented Oreo which is not only scalable but also accurate. Oreo contributes to the state of the art by pushing the boundary of clone detection to the Twilight Zone. Oreo's approach and components provide opportunity to build clone detectors with better recall and precision in the Twilight Zone. One can use concepts from information retrieval to improve the performance of Oreo. For example, concepts like stemming, stop words, and synonyms can be used to improve recall. Analysis of false positives in the output of Oreo reveals that many of these false positives have methods which use too many *asserts* or *append* method calls. This false positives can be avoided by tagging such words as stop words or by assigning weights to different words. Important concepts like $TF-IDF$, could play vital role in assigning weights to Action tokens.

Oreo uses software metrics to find structural similar methods. More metrics can be added to the already existing metrics to compare methods on wider points. Some of these new metrics could focus on the properties of literals like average length of string literals. Such metrics could play important role in improving the precision of the approach. In our approach we trained our machine learning model using a labeled dataset which was produced using SourcererCC. It could happen that Oreo learned characteristics of clone pairs that are more dominant in the output of SourcererCC. A humanly curated dataset containing equal number of instances for different types and structural properties of clone pairs could be beneficial to train a model. This raises a need to curate datasets which can be used for machine learning purposes.

Apart from finding method level clones across and within projects, Oreo can be modified to find clones at other granularities. Our approach can also be modified to be used on languages other than Java. Moreover, finding similar pieces of code written in different language is also possible. Such modifications would require a careful selection of software metrics and also changes in the semantic filter.

Going forward, clone detection techniques should not be limited to detecting clones in source code. These techniques can be used (after modifications) in finding similarity across other artifacts like bug-reports, binary code, and requirement documents among others. New use cases of clone detection such as code search opens up many possibilities like automatic code repair and code completion. Code synthesis is another exciting area of research which can benefit from clone detection.

Bibliography

[1] Copy/paste detector (cpd). `https://pmd.github.io/pmd-6.6.0/pmd_userdocs_cpd.html`. Accessed: 2018-08-23.

[2] Z. &#272;urić and D. Gašević. A source code similarity system for plagiarism detection. *Comput. J.*, 56(1):70–86, Jan. 2013.

[3] J. S. Alghamdi, R. A. Rufai, and S. M. Khan. Oometer: A software quality assurance tool. In *Proceedings of Ninth European Conference on Software Maintenance and Reengineering (CSMR 2005)*, pages 190–191. IEEE, 2005.

[4] Ambient Software Evoluton Group. IJaDataset 2.0. `http://secold.org/projects/seclone`, January 2013.

[5] M. Andersson and P. Vestergren. Object-oriented design quality metrics. 2004.

[6] E. Arisholm and L. C. Briand. Predicting fault-prone components in a Java legacy system. In *Proceedings of the 2006 ACM/IEEE International Symposium on Empirical software Engineering*, pages 8–17. ACM, 2006.

[7] E. Arisholm, L. C. Briand, and M. Fuglerud. Data mining techniques for building fault-proneness models in telecom Java software. In *Proceedings of the 18th IEEE International Symposium on Software Reliability (ISSRE'07)*, pages 215–224. IEEE, 2007.

[8] B. S. Baker. A program for identifying duplicated code. *Computing Science and Statistics*, pages 24–49, 1992.

[9] P. Baldi and Y. Chauvin. Neural networks for fingerprint recognition. *Neural Computation*, 5(3):402–418, 1993.

[10] P. Baldi and P. Sadowski. The dropout learning algorithm. *Artificial intelligence*, 210:78–122, 2014.

[11] V. R. Basili, L. C. Briand, and W. L. Melo. A Validation of Object-Oriented Design Metrics as Quality Indicators. *IEEE Trans. Softw. Eng.*, 22(10):751–761, 1996.

[12] V. R. Basili, G. Caldiera, and H. D. Rombach. The goal question metric approach. *Encyclopedia of software engineering*, 1:528–532, 1994.

[13] V. R. Basili, F. Shull, and F. Lanubile. Building knowledge through families of experiments. *Software Engineering, IEEE Transactions on*, 25(4):456–473, 1999.

[14] H. A. Basit, D. C. Rajapakse, and S. Jarzabek. Beyond templates: a study of clones in the stl and some general implications. In *Proceedings. 27th International Conference on Software Engineering, 2005. ICSE 2005.*, pages 451–459, May 2005.

[15] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier. Clone detection using abstract syntax trees. In *Proceedings. International Conference on Software Maintenance (Cat. No. 98CB36272)*, pages 368–377, Nov 1998.

[16] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier. Clone detection using abstract syntax trees. In *Proceedings of the International Conference on Software Maintenance*, pages 368–377. IEEE, 1998.

[17] F. Beck and S. Diehl. Evaluating the impact of software evolution on software clustering. In *Proceedings of Working Conference on Reverse Engineering (WCRE'10)*. IEEE Computer Society, 2011.

[18] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, and E. Merlo. Comparison and evaluation of clone detection tools. *IEEE Transactions on Software Engineering*, 33(9):577–591, Sept 2007.

[19] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, and E. Merlo. Comparison and evaluation of clone detection tools. *IEEE Transactions on Software Engineering*, 33(9):577–591, Sept 2007.

[20] H. Benestad, B. Anda, and E. Arisholm. Assessing software product maintainability based on class-level structural measures. *Product-Focused Software Process Improvement*, pages 94–111, 2006.

[21] M. Borrego, E. P. Douglas, and C. T. Amelink. Quantitative, qualitative, and mixed research methods in engineering education. *Journal of Engineering education*, 98(1):53–66, 2009.

[22] J. Börstler, M. Nordström, and J. H. Paterson. On the quality of examples in introductory Java textbooks. *ACM Transactions on Computing Education (TOCE)*, 11(1):3, 2011.

[23] L. C. Briand, J. Wüst, J. W. Daly, and D. V. Porter. Exploring the relationships between design measures and software quality in object-oriented systems. *Journal of systems and software*, 51(3):245–273, 2000.

[24] M. Bruntink, A. van Deursen, R. van Engelen, and T. Tourwé. On the use of clone detection for identifying cross cutting concern code. *IEEE Transactions on Software Engineering*, 31(10):804–818, 2005.

[25] B. B. Cambazoglu, A. Catal, and C. Aykanat. Effect of inverted index partitioning schemes on performance of query processing in parallel text retrieval systems. In *Proceedings of International Symposium on Computer and Information Sciences*, pages 717–725. Springer, 2006.

[26] M. Cartwright and M. Shepperd. An empirical investigation of an object-oriented software system. *Software Engineering, IEEE Transactions on*, 26(8):786–796, 2000.

[27] D. Chatterji, J. C. Carver, and N. A. Kraft. Code clones and developer behavior: Results of two surveys of the clone research community. *Empirical Softw. Engg.*, 21(4):1476–1508, Aug. 2016.

[28] K. Chen, P. Liu, and Y. Zhang. Achieving accuracy and scalability simultaneously in detecting application clones on android markets. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 175–186, New York, NY, USA, 2014. ACM.

[29] S. R. Chidamber and C. F. Kemerer. A Metrics Suite for Object Oriented Design. *IEEE Trans. Softw. Eng.*, 20(6):476–493, 1994.

[30] J. Cordy. Comprehending reality - practical barriers to industrial adoption of software maintenance automation. In *Proceedings of International Conference on Program Comprehension*, pages 196–205, 2003.

[31] N. Davey, P. Barson, S. Field, R. Frank, and D. Tansley. The development of a software clone detector. *International Journal of Applied Software Technology*, 1995.

[32] J. Davies, D. M. Germán, M. W. Godfrey, and A. Hindle. Software bertillonage: finding the provenance of an entity. In *MSR*, 2011.

[33] M. de Wit, A. Zaidman, and A. van Deursen. Managing code clones using dynamic change tracking and resolution. In *Proceedings of the 25th International Conference on Software Maintenance (ICSM 2009)*. IEEE Computer Society, 2009.

[34] P. Di Lena, K. Nagata, and P. Baldi. Deep architectures for protein contact map prediction. *Bioinformatics*, 28(19):2449–2457, 2012.

[35] J. L. Donaldson, A.-M. Lancaster, and P. H. Sposato. A plagiarism detection system. In *Proceedings of the Twelfth SIGCSE Technical Symposium on Computer Science Education*, SIGCSE '81, pages 21–25, New York, NY, USA, 1981. ACM.

[36] S. Dreiseitl and L. Ohno-Machado. Logistic regression and artificial neural network classification models: a methodology review. *Journal of biomedical informatics*, 35(5):352–359, 2002.

[37] S. Ducasse, O. Nierstrasz, and M. Rieger. On the effectiveness of clone detection by string matching. *Journal of Software Maintenance and Evolution: Research and Practice*, 18(1):37–58, 2006.

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

[39] K. El Emam, S. Benlarbi, N. Goel, and S. N. Rai. The confounding effect of class size on the validity of object-oriented metrics. *Software Engineering, IEEE Transactions on*, 27(7):630–650, 2001.

[40] R. Elva and G. T. Leavens. Jsctracker: A semantic clone detection tool for java code. Technical report, University of Central Florida, Dept. of EECS, CS division, 2012.

[41] E. Flores, P. Rosso, L. Moreno, and E. Villatoro-Tello. On the detection of source code re-use. In *Proceedings of the Forum for Information Retrieval Evaluation*, FIRE '14, pages 21–30, New York, NY, USA, 2015. ACM.

[42] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 1999.

[43] M. Gabel, L. Jiang, and Z. Su. Scalable detection of semantic clones. In *Proceedings of the 30th International Conference on Software Engineering*, ICSE '08, pages 321–330, New York, NY, USA, 2008. ACM.

[44] D. M. German, M. D. Penta, Y. Gueheneuc, and G. Antoniol. Code siblings: Technical and legal implications of copying code between applications. In *2009 6th IEEE International Working Conference on Mining Software Repositories*, pages 81–90, May 2009.

[45] X. Glorot, A. Bordes, and Y. Bengio. Deep sparse rectifier neural networks. In *JMLR W&CP: Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics (AISTATS 2011)*, Apr. 2011.

[46] N. Gode and J. Harder. Clone stability. In *Software Maintenance and Reengineering (CSMR), 2011 15th European Conference on*, pages 65–74. IEEE, 2011.

[47] N. Göde and R. Koschke. Incremental clone detection. In *13th European Conference on Software Maintenance and Reengineering (CSMR)*, pages 219–228. IEEE, 2009.

[48] N. Göde and R. Koschke. Frequency and risks of changes to clones. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 311–320. ACM, 2011.

[49] A. Goffi, A. Gorla, A. Mattavelli, M. Pezzè, and P. Tonella. Search-based synthesis of equivalent method sequences. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 366–376. ACM, 2014.

[50] G. Gousios and D. Spinellis. Ghtorrent: Github's data from a firehose. In *2012 9th IEEE Working Conference on Mining Software Repositories (MSR)*, pages 12–21, June 2012.

[51] V. Gupta, K. Aggarwal, and Y. Singh. A fuzzy approach for integrated measure of object-oriented software testability. *Journal of Computer Science*, 1(2):276–282, 2005.

[52] N. Göde and R. Koschke. Incremental clone detection. In *2009 13th European Conference on Software Maintenance and Reengineering*, pages 219–228, March 2009.

[53] M. A. Hall. Correlation-based feature selection for machine learning. Technical report, University of Waikato Hamilton, 1999.

[54] M. H. Halstead. *Elements of Software Science (Operating and programming systems series)*. Elsevier Science Inc., 1977.

[55] K. He, X. Zhang, S. Ren, and J. Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *Proceedings of the 2015 IEEE International Conference on Computer Vision*, ICCV '15, pages 1026–1034. IEEE Computer Society, 2015.

[56] A. Hemel and R. Koschke. Reverse engineering variability in source code using clone detection: A case study for linux variants of consumer electronic devices. In *Proceedings of the 2012 19th Working Conference on Reverse Engineering*, WCRE '12, pages 357–366, Washington, DC, USA, 2012. IEEE Computer Society.

[57] K. Herzig, S. Just, A. Rau, and A. Zeller. Classifying code changes and predicting defects using changegenealogies. Technical report, Nov. 2013.

[58] T. K. Ho. Random decision forests. In *Proceedings of the Third International Conference on Document Analysis and Recognition*, volume 1, pages 278–282. IEEE, 1995.

[59] K. Hotta, Y. Sano, Y. Higo, and S. Kusumoto. Is duplicate code more frequently modified than non-duplicate code in software evolution?: an empirical study on open source software. In *Proceedings of the Joint ERCIM Workshop on Software Evolution (EVOL) and International Workshop on Principles of Software Evolution (IWPSE)*, pages 73–82. ACM, 2010.

[60] T. Ishihara, K. Hotta, Y. Higo, H. Igaki, and S. Kusumoto. Inter-project functional clone detection toward building libraries - an empirical study on 13,000 projects. In *2012 19th Working Conference on Reverse Engineering*, pages 387–391, Oct 2012.

[61] M. R. Islam and M. F. Zibran. A comparative study on vulnerabilities in categories of clones and non-cloned code. In *Software Analysis, Evolution, and Reengineering (SANER), 2016 IEEE 23rd International Conference on*, volume 3, pages 8–14. IEEE, 2016.

[62] A. J. Izenman. *Modern multivariate statistical techniques*, volume 1. Springer, 2008.

[63] Jhawk. http://mondego.ics.uci.edu/projects/clonedetection/.

[64] L. Jiang, G. Misherghi, Z. Su, and S. Glondu. Deckard: Scalable and accurate tree-based detection of code clones. In *29th International Conference on Software Engineering (ICSE'07)*, pages 96–105, May 2007.

[65] L. Jiang and Z. Su. Automatic mining of functionally equivalent code fragments via random testing. In *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis*, pages 81–92. ACM, 2009.

[66] J. H. Johnson. Identifying redundancy in source code using fingerprints. In *Proceedings of the 1993 Conference of the Centre for Advanced Studies on Collaborative Research: Software Engineering - Volume 1*, pages 171–183. IBM Press, 1993.

[67] J. H. Johnson. Substring matching for clone detection and change tracking. In *Proceedings of 1994 International Conference on Software Maintanence*, pages 120–126, 1994.

[68] J.R.Cordy. Exploring large-scale system similarity using incremental clone detection and live scatterplots. In *Proceedings of International Conference on Program Comprehension*, 2011.

[69] E. Juergens, F. Deissenboeck, B. Hummel, and S. Wagner. Do code clones matter? In *Proceedings of ICSE*, pages 485–495, 2009.

[70] D. Kafura and G. Reddy. The use of software complexity metrics in software maintenance. *Software Engineering, IEEE Transactions on*, SE-13(3):335–343, March 1987.

[71] T. Kamiya. Agec: An execution-semantic clone detection tool. In *Proceeings of the 21st IEEE International Conference on Program Comprehension (ICPC)*, pages 227–229. IEEE, 2013.

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

[73] C. Kapser and M. Godfrey. ”cloning considered harmful” considered harmful: patterns of cloning in software. *Empirical Software Engineering*, 13(6):645–692, 2008.

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

[75] D. Kawrykow and M. P. Robillard. Improving api usage through automatic detection of redundant code. In *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, ASE '09, pages 111–122, Washington, DC, USA, 2009. IEEE Computer Society.

[76] I. Keivanloo, C. K. Roy, and J. Rilling. Java bytecode clone detection via relaxation on code fingerprint and semantic web reasoning. In *Proceedings of the 6th International Workshop on Software Clones*, pages 36–42. IEEE Press, 2012.

[77] I. Keivanloo, C. K. Roy, and J. Rilling. Sebyte: A semantic clone detection tool for intermediate languages. In *Proceedings of the 20th IEEE International Conference onProgram Comprehension (ICPC)*, pages 247–249. IEEE, 2012.

[78] M. Kim and D. Notkin. Program element matching for multi-version program analyses. In *MSR*, 2006.

[79] M. Kim, V. Sazawal, D. Notkin, and G. Murphy. An empirical study of code clone genealogies. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE-13, pages 187–196, New York, NY, USA, 2005. ACM.

[80] R. Komondoor and S. Horwitz. Effective, automatic procedure extraction. In *11th IEEE International Workshop on Program Comprehension, 2003.*, pages 33–42, May 2003.

[81] K. Kontogiannis. Evaluation experiments on the detection of programming patterns using software metrics. In *Proceedings of the Fourth Working Conference on Reverse Engineering*, pages 44–54. IEEE, 1997.

[82] R. Koschke. Survey of research on software clones. In *Proceedings of Duplication, Redundancy, and Similarity in Software*, 2007.

[83] R. Koschke. *Identifying and Removing Software Clones*, pages 15–36. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.

[84] R. Koschke. Large-scale inter-system clone detection using suffix trees. In *2012 16th European Conference on Software Maintenance and Reengineering*, pages 309–318, March 2012.

[85] R. Koschke, R. Falke, and P. Frenzel. Clone detection using abstract syntax suffix trees. In *Proceedings of the 13th Working Conference on Reverse Engineering*, WCRE '06, pages 253–262, Washington, DC, USA, 2006. IEEE Computer Society.

[86] J. Krinke. Identifying Similar Code with Program Dependence Graphs. In *Proceedings of the Eighth Working Conference on Reverse Engineering (WCRE'01)*, page 301. IEEE Computer Society, 2001.

[87] J. Krinke. A study of consistent and inconsistent changes to code clones. In *Working Conference on Reverse Engineering*, pages 170–178, 2007.

[88] J. Krinke. Is cloned code more stable than non-cloned code? In *Source Code Analysis and Manipulation, 2008 Eighth IEEE International Working Conference on*, pages 57–66. IEEE, 2008.

[89] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25*, pages 1097–1105. Curran Associates, Inc., 2012.

[90] A. Kulkarni and J. Callan. Document allocation policies for selective searching of distributed indexes. In *Proceedings of the 19th ACM International Conference on Information and knowledge Management*, pages 449–458. ACM, 2010.

[91] S. Kullback and R. A. Leibler. On information and sufficiency. *The Annals of Mathematical Statistics*, 22(1):79–86, 1951.

[92] L. Li, H. Feng, W. Zhuang, N. Meng, and B. Ryder. Cclearner: A deep learning-based clone detection approach. In *Proceedings of the 33rd IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 249–260. IEEE, 2017.

[93] W. Li and S. Henry. Object-oriented metrics that predict maintainability. *J. Syst. Softw.*, 23(2):111–122, Nov. 1993.

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

[95] J. Liebeherr, E. R. Omiecinski, and I. F. Akyildiz. The effect of index partitioning schemes on the performance of distributed query processing. *IEEE Transactions on Knowledge and Data Engineering*, 5(3):510–522, 1993.

[96] S. Livieri, Y. Higo, M. Matushita, and K. Inoue. Very-large scale code clone analysis and visualization of open source programs using distributed ccfinder: D-ccfinder. In *Proceedings of 29th International Conference on Software Engineering (ICSE 2007)*, pages 106–115. IEEE, 2007.

[97] C. Lopes, S. Bajracharya, J. Ossher, and P. Baldi. UCI source code data sets, 2010.

[98] C. V. Lopes, P. Maj, P. Martins, V. Saini, D. Yang, J. Zitny, H. Sajnani, and J. Vitek. Déjàvu: A map of code duplicates on github. *Proc. ACM Program. Lang.*, 1(OOPSLA):84:1–84:28, Oct. 2017.

[99] C. V. Lopes, P. Maj, P. Martins, V. Saini, D. Yang, J. Zitny, H. Sajnani, and J. Vitek. Déjàvu: A map of code duplicates on github. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA), October 2017.

[100] C. V. Lopes and J. Ossher. How scale affects structure in Java programs. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 675–694. ACM, 2015.

[101] A. Lozano and M. Wermelinger. Assessing the effect of clones on changeability. In *Software Maintenance, 2008. ICSM 2008. IEEE International Conference on*, pages 227–236. IEEE, 2008.

[102] A. Lozano, M. Wermelinger, and B. Nuseibeh. Evaluating the harmfulness of cloning: a change based experiment. In *Mining Software Repositories*, pages 18–22, 2007.

[103] D. Lucrédio, E. S. de Almeida, and R. P. Fortes. An investigation on the impact of mde on software reuse. In *Software Components Architectures and Reuse (SBCARS), 2012 Sixth Brazilian Symposium on*, pages 101–110. IEEE, 2012.

[104] A. Maven. http://maven.apache.org/.

[105] J. Mayrand, C. Leblanc, and E. Merlo. Experiment on the automatic detection of function clones in a software system using metrics. In *Proceedings of International Conference on Software Maintenance*, page 244, 1996.

[106] T. J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, SE-2(4):308–320, Dec 1976.

[107] M. B. Miles and A. M. Huberman. *Qualitative data analysis: An expanded sourcebook.* sage, 1994.

[108] M. Mondal, C. K. Roy, M. S. Rahman, R. K. Saha, J. Krinke, and K. A. Schneider. Comparative stability of cloned and non-cloned code: An empirical study. In *Proceedings of the 27th Annual ACM Symposium on Applied Computing*, pages 1227–1234. ACM, 2012.

[109] G. Montavon and K.-R. Müller. Better representations: Invariant, disentangled and reusable. In *Neural Networks: Tricks of the Trade*, pages 559–560. Springer, 2012.

[110] A. Mubarak, S. Counsell, and R. Hierons. Does an 80: 20 rule apply to Java coupling? In *Proceedings of the International Conference on Evaluation and Assessment in Software Engineering*, 2009.

[111] A. Mubarak, S. Counsell, and R. M. Hierons. An evolutionary study of fan-in and fan-out metrics in OSS. In *Research Challenges in Information Science (RCIS), 2010 Fourth International Conference on*, pages 473–482. IEEE, 2010.

[112] H. Murakami, Y. Higo, and S. Kusumoto. A dataset of clone references with gaps. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, MSR 2014, pages 412–415, New York, NY, USA, 2014. ACM.

[113] N. Nagappan, T. Ball, and A. Zeller. Mining metrics to predict component failures. In *Proceedings of the 28th international conference on Software engineering.* ACM, 2006.

[114] E. Nasseri, S. Counsell, and M. Shepperd. An empirical study of evolution of inheritance in Java OSS. In *Software Engineering, 2008. ASWEC 2008. 19th Australian Conference on*, pages 269–278. IEEE, 2008.

[115] J. Neter, M. H. Kutner, C. J. Nachtsheim, and W. Wasserman. *Applied Linear Statistical Models*, volume 4. Irwin Chicago, 1996.

[116] L. A. Neubauer. Kamino: Dynamic approach to semantic code clone detection. *Technical Report, Department of Computer Science, Columiba University, CUCS-022-14*, 2015.

[117] J. Ossher, H. Sajnani, and C. V. Lopes. File cloning in open source Java projects: The good, the bad, and the ugly. In *ICSM.* IEEE, 2011.

[118] K. J. Ottenstein. An algorithmic approach to the detection and prevention of plagiarism. *SIGCSE Bull.*, 8(4):30–41, Dec. 1976.

[119] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Commun. ACM*, 15(12):1053–1058, Dec. 1972.

[120] J.-F. Patenaude, E. Merlo, M. Dagenais, and B. Laguë. Extending software quality assessment techniques to java systems. In *Proceedings of Seventh International Workshop on Program Comprehension*, pages 49–56. IEEE, 1999.

[121] F. Rahman, C. Bird, and P. Devanbu. Clones: what is that smell? *Empirical Software Engineering*, 17(4-5):503–530, 2012.

[122] Rajapakse, D. C., and S. Jarzabek. Using server pages to unify clones in web applications: A trade-off analysis. In *Proceedings of International Conference on Software Engineering*, pages 116–126, 2007.

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

[124] P. Refaeilzadeh, L. Tang, and H. Liu. Cross-validation. In *Encyclopedia of database systems*, pages 532–538. Springer, 2009.

[125] M. Rieger. *Effective Clone Detection Without Language Barriers*. PhD thesis, University of Bern, 2005.

[126] C. K. Roy and J. R. Cordy. A survey on software clone detection research. *Technical Report, Queen's University at Kingston*, 2007.

[127] C. K. Roy and J. R. Cordy. Nicad: Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization. In *2008 16th IEEE International Conference on Program Comprehension*, pages 172–181, June 2008.

[128] C. K. Roy and J. R. Cordy. A mutation/injection-based automatic framework for evaluating code clone detection tools. In *Software Testing, Verification and Validation Workshops, 2009. ICSTW'09. International Conference on*, pages 157–166. IEEE, 2009.

[129] C. K. Roy and J. R. Cordy. Near-miss function clones in open source software: an empirical study. *Journal of Software Maintenance and Evolution: Research and Practice*, 22(3):165–189, 2010.

[130] V. Saini, F. Farmahinifarahani, Y. Lu, P. Baldi, and C. Lopes. Mondego/oreo-artifact: Oreo first release, July 2018.

[131] V. Saini, F. Farmahinifarahani, Y. Lu, P. Baldi, and C. Lopes. Oreo: Detection of clones in the twilight zone. In *Proceedings of the 2018 26th ACM SIGSOFT International Symposium on Foundations of Software Engineering (To Appear)*, FSE 2018, New York, NY, USA, 2018. ACM. `https://arxiv.org/abs/1806.05837`.

[132] V. Saini, H. Sajnani, J. Kim, and C. Lopes. Sourcerercc and sourcerercc-i: Tools to detect clones in batch mode and during software development. In *Proceedings of the 38th International Conference on Software Engineering Companion*, ICSE '16, pages 597–600, New York, NY, USA, 2016. ACM.

[133] V. Saini, H. Sajnani, and C. Lopes. Comparing quality metrics for cloned and non cloned java methods: A large scale empirical study. In *Proceedings of the 2016 IEEE International Conference on Software Maintenance and Evolution (ICSME16)*, pages 256–266. IEEE, 2016.

[134] V. Saini, H. Sajnani, and C. Lopes. Cloned and non-cloned java methods: a comparative study. *Empirical Software Engineering*, 23(4):2232–2278, Aug 2018.

[135] H. Sajnani, V. Saini, and C. V. Lopes. A comparative study of bug patterns in Java cloned and non-cloned code. In *Source Code Analysis and Manipulation (SCAM), 2014 IEEE 14th International Working Conference on*, pages 21–30. IEEE, 2014.

[136] H. Sajnani, V. Saini, J. Ossher, and C. Lopes. Is popularity a measure of its quality? an analysis of maven components. In *Proceedings of the 30th Software Maintenance and Evolution(To appear in ICSME 2014)*. IEEE Computer Society, 2014.

[137] H. Sajnani, V. Saini, J. Svajlenko, C. K. Roy, and C. V. Lopes. Sourcerercc: Scaling code clone detection to big-code. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, pages 1157–1168, May 2016.

[138] I. Samoladas, G. Gousios, D. Spinellis, and I. Stamelos. The sqo-oss quality model: Measurement based open source software evaluation. In B. Russo, E. Damiani, S. Hissam, B. Lundell, and G. Succi, editors, *Open Source Development, Communities and Quality*, pages 237–248, Boston, MA, 2008. Springer US.

[139] R. Scandariato and J. Walden. Predicting vulnerable classes in an android application. In *Proceedings of the 4th international workshop on Security measurements and metrics*, pages 11–16. ACM, 2012.

[140] J. Schmidhuber. Deep learning in neural networks: An overview. *Neural Networks*, 61:85–117, 2015.

[141] A. Sheneamer and J. Kalita. Semantic clone detection using machine learning. In *Proceedings of the 15th IEEE International Conference on Machine Learning and Applications (ICMLA)*, pages 1024–1028, 2016.

[142] A. Sheneamer and J. Kalita. A survey of software clone detection techniques. *International Journal of Computer Applications*, 137:1–21, 2016.

[143] M. Shomrat and Y. Feldman. Detecting refactored clones. In G. Castagna, editor, *ECOOP 2013 European Condeference on Object-Oriented Programming*, volume 7920 of *Lecture Notes in Computer Science*, pages 502–526. Springer Berlin Heidelberg, 2013.

[144] R. Socher, Y. Bengio, and C. D. Manning. Deep learning for nlp (without magic). In *Tutorial Abstracts of ACL 2012*, pages 5–5. Association for Computational Linguistics, 2012.

[145] R. Subramanyam and M. S. Krishnan. Empirical analysis of ck metrics for object-oriented design complexity: Implications for software defects. *IEEE Trans. Softw. Eng.*, 29(4):297–310, Apr. 2003.

[146] J. Svajlenko, J. F. Islam, I. Keivanloo, C. K. Roy, and M. M. Mia. Towards a big data curated benchmark of inter-project code clones. In *2014 IEEE International Conference on Software Maintenance and Evolution*, pages 476–480, Sept 2014.

[147] J. Svajlenko, J. F. Islam, I. Keivanloo, C. K. Roy, and M. M. Mia. Towards a big data curated benchmark of inter-project code clones. In *Proceedings of 2014 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 476–480. IEEE, 2014.

[148] J. Svajlenko and C. K. Roy. Evaluating clone detection tools with bigclonebench. In *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 131–140, Sept 2015.

[149] J. Svajlenko and C. K. Roy. Evaluating clone detection tools with bigclonebench. In *Proceedings of the 2015 IEEE International Conference on Software Maintenance and Evolution*, ICSME '15, pages 131–140, 2015.

[150] J. Svajlenko and C. K. Roy. Bigcloneeval: A clone detection tool evaluation framework with bigclonebench. In *Proceedings of 2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 596–600. IEEE, 2016.

[151] J. Svajlenko and C. K. Roy. Bigcloneeval: A clone detection tool evaluation framework with bigclonebench. In *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 596–600, Oct 2016.

[152] J. Svajlenko and C. K. Roy. Cloneworks: a fast and flexible large-scale near-miss clone detection tool. In *Proceedings of the 39th International Conference on Software Engineering Companion*, pages 177–179. IEEE Press, 2017.

[153] J. Svajlenko and C. K. Roy. Fast and flexible large-scale clone detection with cloneworks. In *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*, pages 27–30, May 2017.

[154] R. Tekchandani, R. K. Bhatia, and M. Singh. Semantic code clone detection using parse trees and grammar recovery. In *Confluence 2013: The Next Generation Information Technology Summit*. IET, 2013.

[155] M. S. Uddin, C. K. Roy, and K. A. Schneider. Simcad: An extensible and faster clone detection tool for large scale software systems. In *IEEE 21st International Conference on Program Comprehension (ICPC)*, pages 236–238. IEEE, 2013.

[156] P. Wang, J. Svajlenko, Y. Wu, Y. Xu, and C. K. Roy. Ccaligner: A token based large-gap clone detector. In *Proceedings of the 40th International Conference on Software Engineering*, ICSE '18, pages 1066–1077, New York, NY, USA, 2018. ACM.

[157] T. Wang, M. Harman, Y. Jia, and J. Krinke. Searching for better configurations: a rigorous approach to clone evaluation. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, pages 455–465. ACM, 2013.

[158] H.-H. Wei and M. Li. Supervised deep features for software functional clone detection by exploiting lexical and syntactical information in source code. In *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence (IJCAI-17)*, pages 3034–3040, 2017.

[159] M. White, M. Tufano, C. Vendome, and D. Poshyvanyk. Deep learning code fragments for code clone detection. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ASE 2016, pages 87–98, New York, NY, USA, 2016. ACM.

[160] X. Wu, V. Kumar, J. R. Quinlan, J. Ghosh, Q. Yang, H. Motoda, G. J. McLachlan, A. Ng, B. Liu, S. Y. Philip, et al. Top 10 algorithms in data mining. *Knowledge and information systems*, 14(1):1–37, 2008.

[161] S. Xie, F. Khomh, Y. Zou, and I. Keivanloo. An empirical study on the fault-proneness of clone migration in clone genealogies. In *Proc. of CSMR-WCRE*, pages 94–103. IEEE, 2014.

[162] D. Yang, P. Martins, V. Saini, and C. Lopes. Stack overflow in github: Any snippets there? In *Proceedings of the 14th International Conference on Mining Software Repositories*, MSR '17, pages 280–290, Piscataway, NJ, USA, 2017. IEEE Press.

Appendices

A Experience Report on Using Software Metrics for Clone Detection

Metric-based approaches are not new to detect clones in software systems. Ottenstein in 1976 used Halstead complexity measures to detect plagiarism in software [118]. Since then other researchers have also proposed metrics based approaches to detect similar code in software [35, 81, 105]. Software metrics based approaches show good promise if th goal is to detect very similar code clones, which usually fall into Type I and Type II clone categories. The intuition is that if two code snippets are copy-paste clones, then the metrics computed on these two snippets should be very similar. Therefore, instead of comparing tokens of the two snippets, one can compare these metrics values and identify clones if the absolute difference is less than or equal to the delta threshold defined for each of the corresponding metrics. However, as the structural differences between the snippets start to increase, for example in Type III clones, it becomes difficult to manually configure these delta values for each of the metrics used in clone detection technique. Moreover, different combinations of these delta values are needed to be considered to identify clones with good accuracy, a configuration explosion problem. It becomes very difficult to humanly analyze all different configuration which makes it difficult to use metric based approaches to detect clones with high accuracy. Moreover, metric based approaches also suffer from candidate explosion problem which makes it very difficult for them to scale large datasets. During my work on using software metrics for clone detection I also faced these issues first hand.

My work on comparing cloned methods to non cloned methods [133] seeded the idea to use software metrics for clone detection. I along with Farima, a fellow Ph.D. student, started exploring the idea of using software metrics to reduce the comparisons between the code snippets. We tried an approach where we calculated some simple metrics for every code snippets. These metrics include *number of language tokens*, *number of unique language* *tokens*, *number of characters*, and *number of operators*. We wanted to retrieve the possible clone candidates of a query method in constant time. to this end, we calculated the above metrics for every method in our dataset. Then, we sorted these methods based on the number of tokens in these methods. To get all candidate clones of a query method, we did two binary searches on this sorted dataset to find the upper and lower index of the methods such that the number of tokens in these methods are similar to that of the query method. The idea behind this approach was that two methods in a clone pair should have similar number of tokens. Using this approach, we were successful in filtering out a large number of methods without performing any comparisons between them and the query method. Now, the next challenge was to compare the query method with all candidate methods to identify clone pairs. To this end, we explored with comparing the remaining metrics of every candidate pair. Soon we realized that for each metric comparison we need to manually set the threshold values. We did not know what should be these thresholds. For example, should we say that a pair is a clone pair if their number of tokens are with in 80% and number of unique tokens are within 90% and so on. To get this threshold values we decided to manually look at the clone pairs and their corresponding metric values. After multiple iterations of manually setting the threshold values and computing clone detection using our approach, we learned that our approach is not good in terms of precision.

To overcome the configuration explosion challenge, we decided to use machine learning. To this end, we used the dataset of 3,562 Java projects which I created in one of my previous work [133]. The dataset consists of 3,562 Java projects hosted on Maven [104]. The comprehensive list of projects with their version information is available at `http://mondego.ics.uci.edu/projects/clone-metrics/`.

Dataset Artifacts and Properties: We used following artifacts from the dataset described above:

1) Clone-pairs. Intra-project method-level clone detection results for 3,562 projects. The clone detection was carried out using SourcererCC. Overall, 412,705 cloned and 616,604 non-cloned methods were identified by SourcererCC in this dataset.

2) Method-level metrics. This artifact consists of method-level metrics calculated for the methods in this dataset. The metrics were calculated at the using version 6 of the JHawk tool [63]. JHawk has been widely used in academic studies on Java metrics [3, 6, 7, 20, 51]. Table A.1 shows the 25 metrics for which the computed values were available in the artifact. Many of these metrics are standard metrics. A set of metrics are derived from the Software Quality Observatory for Open Source Software (SQO-OSS) [138]. SQO-OSS is composed of well-established and validated software quality metrics, which can be computed either from source code or from surrounding community data. More details on these metrics and the dataset can be found elsewhere [133].

A.1 Train and Test Data Creation

We separated the whole dataset at hand randomly to two parts: 60% for *train* dataset, and the rest *test*. We produced features for both datasets, and for train dataset, we integrated each Feature Vector with its corresponding *is_clone* label based on SourcererCC.

To have a manageable sized dataset, we filtered out pairs for which the percentage difference in NOS is more than 30%. Having the dataset ready, the first step to train the target clone detector model was to select a classification model which fits our needs and has a satisfactory performance both in terms of accuracy and timing. This process is explained in Section A.1.1. The trained model was then tested on the reserved unseen test data; results of this process is explained in Section A.1.2.

Table A.1: Software Quality Metrics

Name	Description
XMET	External methods called by the method
VREF	Number of variables referenced
VDEC	Number of variables declared
TDN	Total depth of nesting
NOS	Number of statements
NOPR	Total number of operators
NOA	Number of arguments
NLOC	Number of lines of code
NEXP	Number of expressions
NAND	Total number of operands
MOD	Number of modifiers
MDN	Method, Maximum depth of nesting
LOOP	Number of loops (for,while)
LMET	Local methods called by the method
HVOL	Halstead volume
HVOC	Halstead vocabulary of method
HLTH	Halstead length of method
HEFF	Halstead effor to implement a method
HDIF	Halstead difficulty to implement a method
HBUG	Halstead prediction of number of bugs
EXCT	Number of exceptions thrown by the method
EXCR	Number of exceptions referenced by the method
CREF	Number of classes referenced
COMP	McCabes cyclomatic complexity
CAST	Number of class casts

A.1.1 Model Selection

To select a target model, we performed 10-fold cross validation, (a common form of N-fold cross validation [124]), with various algorithms. However, since our training dataset was 45GB in size, training multiple models using 10-fold cross validation(CV) was not feasible in terms of timing. Hence, we selected three different random samples from this dataset: a sample with 10 thousand rows, a sample with 50 thousand rows, and another with 100 thousand rows. Having three different samples could help us get assured that the model selection process is general and independent of characteristics of a single dataset sample. Each of the three sampled datasets were split into 70% train and 30% test. 10-fold cross validation was performed on the training dataset, and then each model was trained on the whole training dataset and tested on testing dataset. The purpose was to reserve an unseen portion in each sample and compare the results of 10-fold cross validation with results on an unseen bunch of data. The results of this 10-fold cross validation is given in the Table A.2.

We ran our experiments with multiple classifiers: K-Nearest Neighbors (KNN) [160], Naive Bayes [160] , Classification and Regression Trees (CART) [160], Logistic Regression (LR) [36], Linear Discriminant Analysis (LDA) [62]. For these models, we measured the Precision and Recall (with respect to SourcererCC) , and also, the time each of them takes. Performance of each classifier after being trained on 70% training dataset and validating on the 30% unseen testing data is given in Table A.3. In this table, $\mathrm{T_L}$ denotes time taken for learning a model, $\mathrm{T_P}$ stands for time taken to predict clones, and time measurements are in seconds. Precision and Recall numbers are measured with respect to the reference clone detector (SourcererCC).

Based on the experiments we selected CART as it had better accuracy and speed than others. However, classification models are prone to get biased and over fitted to train data and not to generalize to unseen data well. As a result, we tested Random Forest [58], which builds several Decision Trees and takes the majority vote among them as the final vote. As it was expected, Random Forest improved accuracy, while taking more time than CART. However, the increase in time was negligible. Hence, Random Forest was selected as the final model. The next step was to select an optimal configuration for Random Forest. To this end, we decided to run experiments on our sampled datasets and then proceed with a select set of configurations to the whole train dataset. After doing 10-fold cross validation, and training on 70% of data and testing on 30% of it, using multiple parameters on the sampled datasets, we found 5 configurations that were performing better than others. Results for these shortlisted configurations on the 100 thousand rows dataset are given in Table A.4.

Having shortlisted 5 classifiers, we selected the configuration presented in first row as the best one. The decision was made on the basis of F1 score and prediction time taken by each model. We then proceeded to train this model on the whole train dataset and predict using it on the unseen 40% reserved portion of data.

A.1.2 Evaluation

To measure the performance of the selected model on unseen data, we decided to train a model on the whole train dataset. However, after analyzing the this dataset, we observed that there exists approximately 7 times more cloned pairs than non-cloned pairs. Hence, we did an inclusive sampling over the data in which equal number of clones and non-clones were present. This ensured us that the model is being trained correctly and accurately. Testing the trained model on the 40% unseen test data, resulted in 88% Precision and 97% Recall with respect to SourcererCC's results. Model train took 570 seconds and Prediction took 345 seconds for the whole test dataset which were promising numbers. These numbers gave us the confidence in using machine learning to address the problem of configuration explosion.

Table A.2: Results of 10-fold Cross Validation on Sampled Rows from Train Dataset

Dataset	10 Thousand Rows				50 Thousand Rows				100 Thousand Rows			
Result / Model	Prec.	Recall	F1	Time	Prec.	Recall	F1	Time	Prec.	Recall	F1	Time
KNN (K=5)	92%	92%	92%	1.6	92%	96%	94%	33.6	93%	97%	95%	145.3
KNN (K=10)	91%	94%	93%	1.5	92%	97%	94%	30.3	93.4%	97%	95%	135.3
Naive Bayes	89%	93%	90%	0.2	90%	92%	91%	1.5	90%	92%	91%	2.6
CART	89%	95%	91%	0.6	92%	96%	94%	3.2	93%	97%	95%	7.2
SVM	96%	89%	93%	31.7	96%	93%	94%	1632	95%	95%	95%	12771
LR	92%	94%	93%	1.1	91%	93%	92%	7.4	92%	93%	92%	18.1
LDA	87%	94%	90%	0.3	88%	94%	91%	1.9	88%	94%	91%	4.6
Random Forest	93%	94%	94%	0.5	94%	96%	95%	2.7	95%	97%	96%	5.7
AdaBoost	93%	93%	94%	2.9	94%	96%	95%	11.7	95%	97%	96%	24.4

Table A.3: Results of Training Models and Validating Them on Unseen Portion of Sampled Rows from Train Dataset

Dataset	10 Thousand Rows					50 Thousand Rows					100 Thousand Rows				
Result / Model	Prec.	Recall	F1	T_L	T_P	Prec.	Recall	F1	T_L	T_P	Prec.	Recall	F1	T_L	T_P
KNN	90%	92%	91%	0.03	0.7	93%	97%	95%	0.2	21	94%	97%	95%	0.7	68.7
Naive Bayes	86%	92%	89%	0.02	0.01	92%	92%	92%	0.1	0.05	90%	91%	91%	0.2	0.08
CART	88%	95%	91%	0.1	0.008	92%	97%	94%	0.3	0.03	93%	96%	95%	0.7	0.05
SVM	93%	89%	91%	3.6	1.4	97%	93%	95%	211	35.4	96%	94%	95%	2079	143
LR	89%	93%	91%	0.1	0.005	92%	93%	93%	0.7	0.03	91%	92%	92%	1.6	0.05
LDA	87%	94%	90%	0.03	0.01	89%	94%	92%	0.2	0.04	87%	93%	90%	0.3	0.06
Random Forest	91%	93%	92%	0.04	0.01	94%	96%	95%	0.2	0.03	95%	96%	95%	0.5	0.09
AdaBoost	89%	94%	91%	0.2	0.01	95%	96%	95%	0.9	0.04	94%	96%	95%	2.9	0.09

Table A.4: Performance of Random Forest Classifier on 100 Thousand Rows Sample

#Est.	Max D	Min S	Min L	Max F	10- Fold Cross Validation				30% Unseen Data (2-Fold Cross Validation)				
					Prec.	Recall	F1	Time	Prec.	Recall	F1	t	t
5	20	10	5	Sqrt(#Features)	95%	96%	95%	5.6	94%	96%	95%	0.43 s	0.06 s
5	–	–	–	–	95%	97%	96%	5.7	95%	96%	95%	0.49 s	0.09 s
10	10	20	5	Sqrt(#Features)	94%	95%	95%	8.1	95%	95%	95%	0.82s	0.084 s
50	5	–	–	Sqrt(#Features)	95%	86%	90%	26.1	95%	85%	90%	2.81 s	0.18 s
25	10	–	–	Sqrt(#Features)	95%	96%	96%	17.5	95%	95%	95%	1.83 s	0.126 s

B Running Oreo

B.1 Prerequisites

B.1.1 System Requirements

You need Linux OS (We have not tested Oreo on any Operating System other than Linux-CentOs), and at least 12 GB of RAM to run Oreo. Also, you need to have Java 8, and Python 3.6 installed. After installing these two, follow the following instructions to download all the prerequisites.

B.1.2 Required Dependencies

Oreo needs that the TensorFlow library for Python is installed; this is needed for running the deep learning part of Oreo. The best way to install TensorFlow and other required dependencies is by creating a virtual environment (venv) for Python.

- Create a virtual environment using the following command:

  $python3 - mvenv/path/to/new/virtual/environment$

- Start the virtual environment:

  $source/path/to/new/virtual/environment/bin/activate$

- Change to $/path/to/new/virtual/environment$, and there, change to

  $oreo/python_scripts/dependencies/$ directory and run the following command to install all the required dependencies:

  $pipinstall - rdependencies.txt$

B.2 Generate Input for Oreo.

Oreo uses a combination of metrics, machine learning, and information retrieval to produce its pairs. Hence, as a preprocessing step, we need to have metrics for the methods in its input ready. Metrics calculation is a one-time process, and when you prepare the metrics for an input dataset ready, you can run the clone detection part on these metrics for as many times as you want. In order to make the execution of this artifact simple, we have generated the needed metrics file for the BigCloneBench, and put it at $https://drive.google.com/open?id=1AUC2uA7ik7ZWrQ4x9ZcKGyyX6YNStG_y$ (The metrics file is for the reduced version that is used by BigCloneEval in recall studies, and we also used it for evaluating Oreo's recall. The name for this file is $blocks.file$. You can skip to Section B.4 if you want to run Oreo with this file. Also, since executing Oreo on the whole dataset of BigCloneBench can take time, we are providing a quick way of running Oreo using a small sample of BigCloneBench dataset (10,000 methods); to run Oreo using this sample, skip to Section B.5. Otherwise, follow the steps below to generate the metrics for your input dataset. To generate the input metrics file, the Metric Calculator tool, which we provide with Oreo, is used. The tool needs to know the absolute path of the dataset for which this input file needs to be created. The path should point to the directory under which the Java files are included. Java files need to be located at the second level of hierarchy in this directory (it means that the path should point to the directory containing some folders where each folder has the Java files). Metrics calculator calculates metrics for the methods found in these Java files, and it creates an output file where each line corresponds to the metrics for one method. This file is then used as the input to Oreo. Follow the following steps to generate the input file:

- In a terminal, go to the root folder of Oreo (*/path/to/new/virtual/environment/oreo*), and then change directory to $java-parser$. Then, run *ant* command to create the needed jar file: *antmetric*

- Then, again change the directory to the root folder of Oreo

  ($/path/to/new/virtual/environment/oreo$), and then change to $python_scripts$ directory; there, you need to run the $metricCalculationWorkManager.py$ script that launches the jar file for metric calculation. This script should be run as follows:

  $python3metricCalculationWorkManager.py1d < absolutepathtoinputdataset >$

  You need to replace $< absolutepathtoinputdataset >$ with the absolute path to the directory containing the input Java source files. As mentioned earlier, Java files need to be located at the second level of hierarchy of the directory whose address is provided. Please note that we are providing the BigCloneBench dataset source files. It is located at $https : //drive.google.com/open?id = 1AUC2uA7ik7ZWrQ4x9ZcKGyyX6YNStG_y$, under the name $bcb_dataset.zip$. You can download this file, unzip it, and provide the path to the unzipped directory to the above command. After issuing the above two commands, look for two files in the current directory: *metric.out* and *metric.err*. Any possible error will be printed in *metric.err* file, and *metric.out* shows the progress of metric calculation process. Once the process is ended, there will be a *done*! printed in the last line of *metric.out* file. It will take about 8 minutes on a system with 4 cores and 32GB of RAM to generate the metrics. When the process ends, you will have a folder named 1_metric_output inside the current directory ($python_scripts$). Inside this folder, there will be a file named $mlcc_input.file$. This file will be used as the input for Oreo.

B.3 Setting Up Oreo

In order to proceed to clone detection step, we need to place metrics file in the appropriate place for Oreo to use it. To this end, follow these steps:

- If you are using the metrics file provided by us : Download $blocks.file$ from this link and place it inside

$/path/to/new/virtual/environment/oreo/clone-detector/input/dataset/$

- If you are using the metrics file that you generated in Section B.2: First, copy the generated $mlcc_input.file$ file to this path:

  $/path/to/new/virtual/environment/oreo/clone-detector/input/dataset/$, and then, rename it to $blocks.file$. Please make sure that there is no other file present at this location.

B.4 Running Oreo

After completing the setup, follow the following steps to run Oreo clone detection. Note that we have run the clone detection process on a Linux system , with Intel(R) Core(TM) i5-4670 CPU @ 3.40GHz having 4 cores, using 12 GB of RAM, and it took 65 minutes to complete the detection process.

- Change the directory to the root of Oreo, and then to $clone-detector$. There, run the following command:

  $pythoncontroller.py1$

- Now, open another terminal and change directory to

  $/path/to/new/virtual/environment/oreo/python_scripts.$

  Then, run the following command:

  $./runPredictor.sh$

  Once the clone detection process is finished, the clone pairs will be reported in several $.txt$ files located at

  $/path/to/new/virtual/environment/oreo/results/predictions/$

  You need to concatenate them in one file to have them all in one place. Please note that each time you want to run Oreo after one round of execution, you need to run

$./cleanup.sh$ before $pythoncontroller.py1$.

B.5 Quick Execution of Oreo

In order to make the execution of Oreo simpler and quicker, we are providing a smaller version of input metrics file so that readers can go through the execution of Oreo faster. This file is located at the $example_input/$ folder of repository, and it is named $blocks_small.file$. In order to do the fast execution, copy this file to $/path/to/new/virtual/environment/oreo/clone-detector/input/dataset/$, and rename the file to $blocks.file$. Then, follow the instructions in Section B.4 to run Oreo on this input. Please note that no other file is placed in this location. It would normally take about 2 minutes for this input to complete the clone detection.