UNIVERSITY OF CALIFORNIA,
IRVINE

Functional Clone Detection in Intelligent Software Components

DISSERTATION

submitted in partial satisfaction of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

in Software Engineering

by

Farima Farmahinifarahani

Dissertation Committee:
Professor Cristina V. Lopes, Chair
Professor Sam Malek
Assistant Professor Iftekhar Ahmed

2022

DEDICATION

To my loving mom.

TABLE OF CONTENTS

Page

LIST OF FIGURES **vi**

LIST OF TABLES **vii**

LIST OF ALGORITHMS **viii**

ACKNOWLEDGMENTS **ix**

VITA **xi**

ABSTRACT OF THE DISSERTATION **xiii**

1 Introduction **1**
- 1.1 Background . . . 1
- 1.2 Motivation . . . 3
- 1.3 Problem Statement . . . 4
- 1.4 Thesis . . . 5
- 1.5 Contributions . . . 7

2 Related Work **9**
- 2.1 Code Clone Detection . . . 9
- 2.2 Similarity Detection of DNN Models . . . 12
- 2.3 Fuzz Testing Neural Networks . . . 14

3 Code Clone Detection **16**
- 3.1 Detecting Clones with Oreo . . . 17
  - 3.1.1 Introduction . . . 17
  - 3.1.2 Reproduction Study . . . 21
  - 3.1.3 The Oreo Clone Detector . . . 29
  - 3.1.4 Learning Metrics . . . 33
  - 3.1.5 Evaluation of Oreo . . . 36
  - 3.1.6 Manual Analysis of Semantic Clones . . . 40
  - 3.1.7 Limitations of this Study . . . 43
  - 3.1.8 Conclusions and Future Work . . . 43
- 3.2 On Precision of Clone Detectors . . . 44
  - 3.2.1 Introduction . . . 44

3.2.2 Study Design . . . 47
3.2.3 Undifferentiated Precision Experiments . . . 50
3.2.4 Type-based Precision Experiments . . . 52
3.2.5 Qualitative Analysis . . . 61
3.2.6 Related Work . . . 66
3.2.7 Threats to Validity . . . 69
3.2.8 Conclusions and Future Work . . . 70

4 Similarity Detection of Neural Network Models with RICA 72
4.1 Introduction . . . 72
4.2 Problem Definition . . . 73
4.2.1 Functional Similarity . . . 74
4.2.2 Challenges of DNN Clone Detection . . . 75
4.3 Goals and Scope . . . 76
4.4 Design of RICA . . . 78
4.4.1 Similarity Inspection Pipeline . . . 78
4.4.2 Functional Similarity Metrics . . . 81
4.4.3 Similarity Thresholds . . . 83
4.5 Balanced Random Inputs . . . 89
4.5.1 Algorithm . . . 91
4.5.2 Parameter Tuning . . . 93

5 Experimental Evaluation of RICA 95
5.1 D56K Evaluation Dataset . . . 96
5.2 Accuracy vs. Models' Similarity . . . 96
5.2.1 Methodology . . . 96
5.2.2 Results . . . 98
5.3 Unknown Models . . . 110
5.3.1 Overall Similarity Results . . . 110
5.3.2 Manual Analysis . . . 112
5.3.3 Discussion . . . 116
5.4 Similarity Analysis of Regression Models . . . 117
5.4.1 Dataset . . . 117
5.4.2 Experiments . . . 117
5.5 Threats to Validity . . . 120

6 Analytical Discussions on RICA 121
6.1 Sensitivity Analysis . . . 122
6.1.1 Sensitivity Analysis Dataset (D14) . . . 122
6.1.2 Sensitivity Analysis Results . . . 124
6.2 Taxonomy of DNN Clones . . . 129

7 Conclusions 131
7.1 Dissertation Summary . . . 131
7.2 Future Work . . . 133

Bibliography **134**

LIST OF FIGURES

Page

3.1 Overview of Oreo. 28
3.2 The Trained Siamese Architecture Model 35
3.3 Overview of the Precision Study 47
3.4 Voting on a Clone Pair 47
3.5 Distribution of Tools' Clone Pairs Per Clone Type 54
3.6 Tools' Precision: Undifferentiated and Type-based 59

4.1 Overview of the similarity detection pipeline by RICA 79
4.2 Overlap metric: random unconstrained inputs 86
4.3 Frequency of labels with Unconstrained Random Inputs 87
4.4 Overlap metric: balanced inputs 88
4.5 Frequency of labels with Balanced Inputs by BRINC 90

5.1 BRINC Generated Image Example 98
5.2 Similarity vs. Accuracy: MN Ver1 same shape as reference model 99
5.3 Similarity vs. Accuracy: MN Rev Clr same shape as reference model 100
5.4 Similarity vs. Accuracy: FMN same shape as reference model 101
5.5 Similarity vs. Accuracy: Iris same shape as reference model 102
5.6 Similarity vs. Accuracy: Sonar same shape as reference model 103
5.7 Similarity vs. Accuracy: MN Ver1 compatible shape as reference model . . . 104
5.8 Similarity vs. Accuracy: FMN compatible shape as reference model 105
5.9 Similarity vs. Accuracy: CNN compatible shape as reference model 106
5.10 Similarity values for GitHub models 112
5.11 Similarity values for Raw Data Model 114
5.12 Similarity values for No Data Model 115
5.13 Similarity values vs. R^2 for Boston housing prices models 120

6.1 Similarity with MN Ver1. Each box: minimum, 1^{st} quartile, median, 3^{rd} quartile, maximum. 125

LIST OF TABLES

Page

3.1 Software Quality Metrics 24
3.2 Results of 10-fold Cross Validation on the Sampled Rows of the Training Dataset 26
3.3 Results of Models Training and Prediction on the Hold-out Testing Part of the Sampled Rows of Dataset 26
3.4 Different Configurations of Random Forest Classifier on 100 Thousand Rows Sample 27
3.5 Final Method-Level Software Metrics Used in Oreo 33
3.6 Recall and Precision Measurements on BigCloneBench 38
3.7 Clone Detection Tools' Characteristics and Configurations 49
3.8 Undifferentiated Precision and Differentiated Recall 52
3.9 Number of Clone Pairs Reported by Each Tool Per Type 52
3.10 Type-Based Precision 59
3.11 Precision: Majority Vote vs. Unanimous Vote 61

4.1 Summary of BRINC's parameters. 94

5.1 RICA's successes and errors per similarity metric 108
5.2 RICA's Precision, recall, accuracy per similarity metric 108
5.3 Similarity Statistics for GitHub Models 112

6.1 Sensitivity analysis: trained models (D14 dataset) 123
6.2 Similarity between all models under study using the three metrics. 126

LIST OF ALGORITHMS

		Page
1	Algorithmic implementation of Type I classification	56
2	Algorithmic implementation of Type II classification	58
3	Output compatibility	80
4	Input compatibility	80
5	Mutate an input	91
6	Generate balanced random inputs	93

ACKNOWLEDGMENTS

First and foremost, I would like to thank my advisor, Professor Cristina Lopes for her continuous support during my PhD studies. Crista always listened to my ideas, brainstormed with me, and helped me turn my ideas into research projects and solutions. She was very hands-on in helping me during my research endeavors, and at the same time, gave me the freedom to explore different ideas and trained me to be an independent researcher. I am forever grateful for her patience, support and help.

I am also thankful to my committee members, Professor Sam Malek and Professor Iftekhar Ahmed, for agreeing to devote their time to be in my committee and for their valuable comments and feedback on my PhD work.

I would also like to thank all my labmates with whom I collaborated during my time at UCI. Specially, I would like to thank Dr. Vaibhav Saini, who was a senior member of the lab when I joined and served as my second mentor. I learned a lot from Vaibhav and am thankful to him for his support and guidance whenever I needed, even till now. I would also like to thank Dr. Hitesh Sajnani for his guidance and help both in my research and my career path.

I would like to express my gratitude to everyone I worked with during my research internships, especially to Dr. Suresh Thummalapenta, Dr. Anirudh Santhiar, Sina Jafari, and Dr. Vaibhav Saini. Working with them gave me valuable insights on the industrial aspects of research.

I am beyond thankful to my mom for her unconditional love and support. She has made tremendous sacrifices throughout her life, so that I be in the place that I am today. Although being far from each other for almost six years, she has always been there for me, any time of the day or night, to comfort me, give me confidence, listen to my worries, and make me feel loved and cared. Words cannot express how much I am thankful to her for believing in me, understanding me, and loving me the way she does. I would also like to thank my grandparents, for their constant love since my childhood till now, and for being beside my mom in my absence. They are the definition of love and kindness to me. I am also thankful to my friends Mahsa, Pantea and Parastoo for cheering me up in times of sorrow and for giving me the confidence to carry on. Finally, I would like to thank Hormoz for being the person to rely on in all steps of my doctorate journey. He supported me during the tough nights of paper deadlines and exams, and talked to me whenever I had doubts or worries. He listened to all my ideas and brainstormed with me although they weren't in his area of expertise. I am very much grateful for his patience and for being the person to turn into whenever I faced a hurdle.

I was fortunate to be supported by the department of Informatics at the University of California, Irvine and by grants from the National Science Foundation during my PhD studies.

I am grateful to the following publishers for including some of my previous work in this

dissertation. The material in Chapter 3.1 is included with the permission of the ACM and is based on the work in:

- Saini, V., Farmahinifarahani, F., Lu, Y., Baldi, P., Lopes, C., "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), 2018, pp. 354-365. DOI: https://doi.org/10.1145/3236024.3236026

The material presented in Chapter 3.2 is included with the permission of IEEE and is from the following paper:

- ©2019 IEEE. Reprinted, with permission, from Farmahinifarahani, F., Saini, V., Yang, D., Sajnani, H., Lopes, C., "On Precision of Code Clone Detection Tools", in Proceedings of the 26th IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER'19), 2019, pp. 84-94. DOI: 10.1109/SANER.2019.8668015

In reference to IEEE copyrighted material which is used with permission in this thesis, the IEEE does not endorse any of University of California, Irvine's products or services. Internal or personal use of this material is permitted. If interested in reprinting/republishing IEEE copyrighted material for advertising or promotional purposes or for creating new collective works for resale or redistribution, please go to `http://www.ieee.org/publications_standards/publications/rights/rights_link.html` to learn how to obtain a License from RightsLink. If applicable, University Microfilms and/or ProQuest Library, or the Archives of Canada may supply single copies of the dissertation.

VITA

Farima Farmahinifarahani

EDUCATION

Doctor of Philosophy in Software Engineering — **2022**
University of California, Irvine — *Irvine, CA*

Master of Science in Computer Engineering, Software — **2014**
Sharif University of Technology — *Tehran, Iran*

Bachelor of Science in Computer Software Engineering — **2011**
Azad University, Central Tehran Branch — *Tehran, Iran*

RESEARCH EXPERIENCE

Graduate Research Assistant — **2016–2022**
University of California, Irvine — *Irvine, California*

Research Intern — **Summer 2020**
Microsoft — *Remote*

Research Intern — **Summer 2019**
Microsoft — *Redmond, WA*

TEACHING EXPERIENCE

Teaching Assistant — **2018–2021**
University of California, Irvine — *Irvine, California*

Teaching Assistant — **2016–2017**
University of California, Irvine — *Irvine, California*

REFEREED CONFERENCE PUBLICATIONS

• Farmahinifarahani, F., Lu, Y., Saini, V., Baldi, P., Lopes, C., "D-REX: Static Detection of Relevant Runtime Exceptions with Location Aware Transformer", in Proceedings of the 21st IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM'21), 2021, pp. 198-208.

• Mehta, S., Farmahinifarahani, F., Bhagwan, R., Guptha, S., Jafari, S., Kumar, R., Saini, V., and Santhiar, A., "Data-Driven Test Selection at Scale", in Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE'21), 2021, pp. 1225-1235.

• Klotzman, V., Farmahinifarahani, F., Lopes, C., "Public Software Development Activity During the Pandemic" To Appear in Proceedings of the 15th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM'21).

• Saini, V., Farmahinifarahani, F., Lu, Y., Yang, D., Martins, P., Sajnani H., Baldi, P., Lopes, C., "Towards Automating Precision Studies of Clone Detectors", in Proceedings of the 41st ACM/IEEE International Conference on Software Engineering (ICSE'19), 2019, pp. 49-59 (Distinguished Artifact Award Winner).

• Farmahinifarahani, F., Saini, V., Yang, D., Sajnani, H., Lopes, C., "On Precision of Code Clone Detection Tools", in Proceedings of the 26th IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER'19), 2019, pp. 84-94.

• Saini, V., Farmahinifarahani, F., Lu, Y., Baldi, P., Lopes, C., "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), 2018, pp. 354-365 (Distinguished Paper Award Winner).

Book Chapters

• Saini, V., Farmahinifarahani, F., Sajnani, H., Lopes, C., "Oreo: Scaling Clone Detection Beyond Near-Miss Clones." In Code Clone Analysis, pp. 63-74. Springer, Singapore, 2021.

ABSTRACT OF THE DISSERTATION

Functional Clone Detection in Intelligent Software Components

By

Farima Farmahinifarahani

Doctor of Philosophy in Software Engineering

University of California, Irvine, 2022

Professor Cristina V. Lopes, Chair

Similarity detection in software systems, also known as clone detection, has been a focus of software engineering in the past years. Cloning can be defined based on simple similarity concepts, such as similarity in pure syntactic features, or based on more sophisticated notions of similarity, where the ultimate functionality (or behavior), with less or no syntactic signals, is the focus. The latter type of clones are known as *functional* clones.

A special case of functional clones, which is becoming more prevalent with the advances in artificial intelligence, is concerned with the behavioral similarity among deep neural network (DNN) models. DNN models are *functions* as they define a portion of a software system's functionality. The wide adoption of these components in software brings new challenges to similarity detection techniques. DNN models are black boxes containing matrices of numbers learned from a training dataset. The training code contains little knowledge about the ultimate model's behavior, and similar training scripts and network architectures may end up producing completely different models. Model comparison, therefore, cannot rely on the structural properties of the models or their training scripts. Instead, it must compare the models' outputs on canonical inputs which generally are the training or testing datasets. However, such datasets may not always be available due to reasons such as the independence of models' deployment from their datasets, and privacy or security concerns. These issues

motivate the need for an approach that can automatically detect functional similarity among DNN models in the absence of canonical inputs.

This dissertation starts by looking into the problem of functional clone detection by presenting Oreo, a code clone detection tool focused on clones with diminished syntactic but high semantic similarity. It then presents a systematic study on precision of code clone detection tools, highlighting the importance of looking into clone types (and more importantly, types with decreased syntactic similarity) when measuring the precision. The dissertation continues by formulating the problem of DNN functional similarity detection in the absence of canonical inputs. To solve this problem, it then introduces RICA, a technique that works by generating random inputs to be used instead of canonical inputs for the purpose of similarity detection. Three similarity metrics that can be used with RICA are presented and their strengths and weaknesses are highlighted.

The evaluation of RICA is done by performing extensive experiments using a dataset of more than 56K classifiers collected from GitHub. RICA's evaluation shows that it has high precision and recall, and highlights the effectiveness of the similarity metrics used by it. Running RICA on the entire dataset of 56K classifiers results in performing more than 7 million comparisons and finding a cloning percentage of 26% among the analyzed models. This is followed by showing how RICA's applicability can be extended beyond classifiers: applying RICA on a regression task demonstrates its effectiveness in finding clones of regression models. Furthermore, a sensitivity analysis reveals how certain model and training properties affect the performance of RICA. Finally, a taxonomy of DNN clone types is proposed which helps in specifying the ultimate capabilities and limitations of RICA, as well as being helpful in future studies of DNN similarity detection.

Chapter 1

Introduction

1.1 Background

Software similarity, or clone, detection is the process of finding similar software pieces, and it has been an active research area in the past years [105, 115, 35]. Clones exist in a variety of software artifacts including the associated software documents [57, 71], software data [45], and the most common form of clones: code clones [107, 111, 104, 124]. Code clone detection has various applications including refactoring and rearchitecting [149], code reduction by elimination of unintentional duplicates [78, 113], plagiarism detection [23], copyright and license violation detection [132, 47], and detecting similar software systems, for example, similar malicious mobile applications [132, 25, 97]. The target similarity can be based on simple syntactic features, or based on semantic (or functional) characteristics, or both. In order to classify clones based on the level of syntactic or semantic similarity they exhibit, there have been four types of clone types defined in the literature [18, 102]:

Type I: Identical code fragments except for differences in white space, layout, and comments.

Type II: Identical code fragments except for differences in identifiers and literal values, in addition to Type I variations.

Type III: Similar code fragments with modifications at statement level. Statements can be changed, added or removed, in addition to Type I and Type II differences.

Type IV: Code fragments that perform the same functionality but are different in syntax.

Among the clone types defined, Type I and Type II are the most straight-forward types to detect as they are concerned with syntactical similarities that can be distinguished by simple static analysis of code (using methods such as text or token matching). Starting from Type III, and towards Type IV clones, as the syntactic similarity fades and semantic similarity is emphasized, detection of clones becomes more and more complicated. This class of clones are also often called *functional* or *behavioral* clones. Functional clone detection is crucial to the process of software engineering and maintenance due to its important application scenarios: code optimization and refactoring based on code functionality, detection of similar malicious software agents, and copy-right or plagiarism detection when syntactic signals are omitted.

A more recent type of functional clones are *deep neural network (DNN) functional clones*. The prevalence of DNN models in a diverse set of application domains and their promising results, have made them a suitable choice for many problems and have subsequently, resulted in them being employed in a wide range software systems [22, 30, 101, 70, 141, 135, 140]. DNN models are considered *functions*, as they define a portion of the functionality of the software systems that include them. Functional clone detection, concerned with detecting similarity in the functions of software systems at different granularity levels, hence, needs to take these models into account. This introduces the concept of *DNN functional clone detection*.

1.2 Motivation

Motivating factors to perform functional clone detection for DNN models are in many ways similar to those of the conventional functional clone detection. Here, I detail some of the main motivating factors to perform functional clone detection for DNN models.

- **Model Theft.** As reported in the literature [17, 137], software theft is a pervasive problem that costs billions of dollars each year. One common scenario of software theft is when employees leave companies and take copies of the proprietary software code or data with them, a scenario that is on the rise at an alarming rate [4]. With recent advances in AI and machine learning, and the replacement of traditional software components with DNN models, software is not just computer instructions anymore. Models trained on proprietary data are considered intellectual property of companies and similar to code, if stolen, can impose serious costs. Reports indicate that model theft is starting to occur, too, both by direct copy of the model file that may be slightly modified later [5] and by duplication of the model's functionality using model extraction attacks [130, 100, 58].

  In any of the above cases, theft investigation requires the examination and comparison of models, since those may be the only artifacts available for comparison – the training data most of the times is not packaged for deployment, and may not even be available for analysis for a variety of reasons (such as privacy or security concerns, or simply, the size of data). In fact, when presented with two large systems for comparison, or a comparison of one system against a large base of software artifacts, an investigator of software theft may not even know what kind of functions the included machine learning models implement, and would need to perform the comparison without relying on any extra information. Automatic DNN model clone detection is an important asset in these scenarios.

- **Malware in App Stores.** Another application of DNN model clone detection is in app stores that scan and monitor the submitted apps for malicious components (or agents). The number of mobile apps that use DNN models to support their main features is on the rise: a study in 2019 [146] showed that 81% of the apps that included DNN models, used these models to support their core features. With the ongoing advances in neural networks and the offering of several light-weight machine learning frameworks suitable for mobile apps, it is expected that this trend continues. Traditional program analysis will soon be insufficient to detect malicious behavior as it cannot analyze models' functions. Model comparison needs to be performed to reveal the functional similarities among DNN functions of the submitted apps.

- **Model Search.** Another use-case of DNN models comparisons is in *model search*. An example of this is in searching for a model that performs the same or a similar functionality as to the model at hand, but has a better accuracy or generalization ability. This scenario can, for example, happen when searching for a pre-trained model and customizing it to one's specific needs by performing further training over it (similar to transfer learning). The benefits of training an already trained model (instead of starting from scratch) are easing the development and training of machine learning models in the development of AI-enabled software, reducing the time and computational costs of model development and training, improving the performance and capabilities of the final model, and overcoming insufficient training data.

1.3 Problem Statement

DNN models are created as the result of a training process where a training script defines the model's structure and training parameters. This process consists of running the training script on a training dataset, which is often provided as a separate file. The result of this

process is the model, consisting of a set of weight and bias parameters, in the form of matrices, learned from the training data. These matrices define what the model does given input data. The function of a model, therefore, comes largely from its training data [11]; given different data, similar setup scripts and network architectures may end up producing completely different models [128]. The training code (which may or may not be accompanied with the deployed model) does not include much knowledge about the model's functionality, and cannot be used for comparing models' functions. The ultimate functionality learned by the model is encapsulated in the model itself, making it the suitable target for any model analysis or comparison.

DNN models, however, are often considered to be inscrutable black boxes, due to them being matrices of numbers: their structure does not disclose any insights on the functions being implemented, and identical training runs can result in producing very different numbers. A static analysis of these matrices, therefore, does not reveal much knowledge about models' behavior. As a result, instead of analyzing the internals of DNN models, models' functional clone detection must compare the models' outputs on canonical inputs which generally are the training or testing datasets. Given a canonical set of test inputs, when the outputs of two models are sufficiently similar, then they are similar. However, such datasets may not always be available due to reasons such as the independence of models' deployment from their datasets, and privacy or security issues. This arises the need for finding suitable substitutes to canonical inputs that can be be used for the purpose of similarity detection of DNN models.

1.4 Thesis

The problem formulated above motivates the need for an approach that can detect functional similarity among DNN models in the absence of canonical inputs. To address this need, in

this dissertation, I present RICA (Random Inputs and Correlation Analysis), an automated technique to detect functional clones of DNN models with no need to have access to models' training/testing data or training scripts. RICA is inspired by research on finding functional clones of software code with input/output analysis [52] as well as research on fuzz testing of neural networks [88, 144, 92] in that it uses random inputs (i.e. white noise) as test sets to assess functional similarity. RICA generates random inputs as a substitute for canonical inputs and performs correlation analysis on models' outputs over these inputs to reason about the degree of similarity. The correlation analysis methods serve as RICA's similarity metrics and there are three metrics that can be used with RICA: *Spearman rank correlation, CCA (Canonical Correlation Analysis)*, and *Overlap*. The latter is only applicable in the case of classifiers since it counts the number of overlaps in predictions.

To show the effectiveness of RICA, I present systematic experiments using a dataset of 56,355 classifiers collected from GitHub. The findings of these experiments indicate that random inputs are perfectly suitable for similarity detection. The three metrics used by RICA produce very good results, all of them above 76% accuracy in the detection of similar/dissimilar models. Spearman rank correlation stands out as the most accurate within the scope of this study, with 94% accuracy. CCA is the most general metric, even more than Spearman correlation, as it can be used to compare any two models with any number of outputs; but within our study, CCA has the highest number of erroneous similarity classifications. The simple overlap metric has the highest uncertainty, but it is able to detect similar models that the other two metrics failed to detect. Based on this, my thesis statement is as follows:

RICA is the first approach capable of finding functional similarities among DNN models in the absence of canonical test inputs and without any knowledge about these models' training datasets and training scripts or parameters. The three similarity metrics used by RICA are highly accurate as attested by extensive empirical results.

1.5 Contributions

The key contributions of this dissertion are as follows:

- Presenting Oreo, a joint work with Vaibhav Saini, as a method to detect code clones with decreased syntactic similarity (first part of Chapter 3).
- Presenting a systematic study on precision of code clone detection tools, highlighting the importance of type-based precision studies and asserting the significance of Type III and beyond clones in studying code clone detectors' precision (second part of Chapter 3).
- Formulating the problem of finding functional clones of DNN models, and operationalizing model similarity detection (Chapter 4).
- Presenting RICA as an approach for detecting functional clones of DNN models in the absence of canonical inputs. RICA generates random inputs that can be used in lieu of meaningful canonical inputs for the purposes of finding similar models. We also present a method for generating balance random inputs for classifiers to be used with the Overlap metric (Chapter 4).
- Curating a dataset of more than 56,000 compiled Keras classifiers from GitHub, clustered based on their input and output shapes. This dataset can be used not just for reproducing this work but also for further studies of DNN models (Chapter 5).
- Presenting systematic experiments to evaluate RICA and showing its effectiveness in finding functionally similar DNN classifiers. These experiments unveil the strengths and weaknesses of three similarity metrics used by RICA, being that the Spearman rank correlation is the most accurate metric for detecting functional similarity of classifiers (Chapter 5).

- Applying RICA on a regression task and presenting its similarity detection results which demonstrates how RICA's effectiveness can go beyond classification problems (Chapter 5).

- Presenting a set of sensitivity analysis experiments aimed at showing how various models' characteristics can affect the similarity predictions made by RICA (Chapter 6).

Chapter 2

Related Work

The contributions of this dissertation are related to and inspired by the prior work in code clone detection and similarity detection for neural network models, as well as the work in fuzz testing neural network models. Here, I overview the related work in each of these three areas.

2.1 Code Clone Detection

There has been a large body of work on code clone detection, focused on different ranges of syntactic and semantic clones. Based on the main approach followed by these tools, they can be categorized into text-based, token-based, tree-based, metrics-based, graph-based, learning-based, and dynamic approaches. Here I review some of the prominent tools in each of these categories. A more detailed review is available in [106].

- **Text-based.** Johnson's 1993 work [54] is an approach for identifying exact repetitions in source code using fingerprints, and their 1994 work [55] is an approach for

detecting the text shared among source files using substring matching. Ducasse et al. [33] proposed a language independent approach for code clone detection. Another notable work is NiCad [104] which works by extracting clone blocks, deriving pretty printed and normalized versions of blocks, and then filtering them. Next, potential clones are compared using Longest Common Subsequence algorithm, and clone pairs are identified.

- **Token-based.** Baker [12, 13] proposed dup as an approach for finding duplicate and near-duplicate code in large software systems. CCFinder [61] performs source transformation accompanied by token-based comparison to located clones in a variety of languages. SourcererCC [111] and CloneWorks [126] are two more recent approaches that have shown a good accuracy in detecting Type 1 to Type 3 clones. They both use a hybrid of Token and Index based techniques to find clone pairs.

- **Tree-based.** Yang's work [148] makes use of languages' grammar to pinpoint to compare source codes. Baxter et al. [16] propose an approach for the detection of both exact and near-miss clones using abstract syntax trees applied on C code. Koschke et al. [69] use suffix trees to find syntactic clones with linear time complexity and using linear space. Deckard [51] works by computing characteristic vectors that can approximate the structural information of AST, and then clustering similar vectors using Locality Sensitive Hashing.

- **Metrics-based.** Techniques in this group calculate several metrics for code-blocks, and instead of comparing code blocks directly, use these metrics for the purpose of clone detection. Keivanloo et al. [63] introduced a hybrid metric-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. [83] 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. [91] to find method level clones in Java projects. Kontogiannis et al. [67] used five modified versions of well known metrics that capture data and control flow properties of programs 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.

- **Graph-based.** Krinke [72] proposed an approach for identifying similar code by finding similar subgraphs in the program dependency graph of code. Komondoor and Horwitz's approach [66] performs clone detection by finding isomorphic program dependency subgraphs with the use of program slicing. Gabel et al. [37] augmented Deckard [51] with with semantic information derived from Program Dependence Graphs (PDGs) to find semantic clones on C code. GPLAG [81] is a plagiarism detection tool that works by mining program dependency graphs. The reason for using PDGs in this context is listed as the invariance of PDGs through plagarism. Chen et al. [26] propose an approach for similarity detection among Android apps. They use *centroid*, a geometry characteristic of program dependency graphs, to detect the similarities between code fragments of Android apps.

- **Learning-based.** White et al. [140] present a deep learning-based approach to detect clones at method and file levels. This is done by automatic learning of discriminating features of source code. Wei and Li [138] propose an end-to-end learning approach using Long-Short-Term-Memory (LSTM) network to learn representations of Sheneamer and Kalita [114] use features extracted from programs' ASTs to detect syntactic clones and features extracted from PDGs are to detect semantic clones. These features are then

used in classification to find the clones. CCLearner [79] extracts tokens from cloned and non-cloned methods, and then trains a classifier using this information that can distinguish cloned methods.

- **Dynamic.** Jiang et al. [52] proposed a method to detect semantic clones by generating random inputs for extracted code each fragment based on the input variables used in the fragment. Then, each code fragment is executed with same input values, and whenever the outputs of two code fragments on the same input differs from each other, they are separated into two different clusters. At last, clusters of functionally equivalent code fragments are remained.

2.2 Similarity Detection of DNN Models

Another very related line of work is the study of similarities of neural networks. A large body of work focuses on finding similarities in the representations that these models learn in different layers.

Laakso et al. [76] propose to compare the neural networks' representation by comparing the distances among neural activations. Using this method, they demonstrate that different neural networks trained by back-propagation on the same categorization task, even with different representational encodings of the input patterns and different numbers of hidden units, reach states in which representations at the hidden units are similar.

Raghu et al. [94] propose to use Canonical Correlation Analysis (CCA) for measuring the similarity between neural network representations, and based on that, propose a technique call Singular Vector CCA (SVCCA). Morcos et al. [86] argue that CCA cannot distinguish between noise and signal in the representations and propose projection weighted CCA to mitigate this. They also discuss a number of related findings including that networks that

generalize tend to converge to more similar solutions compared to those that do not generalize, and that wide networks converge to more similar solutions compared to the narrower ones. Kornblith et al. [68] discuss that a method like CCA which is invariant to invertible linear transformation cannot measure meaningful similarities between the representations of higher than the number of datapoints. They propose a similarity index, named CKA, that can be used to measure and compare the similarities of representations within and across neural networks and does not suffer from CCA's problems. They also show that models learned from different datasets can have similar representations at their early layers.

Li et al. [80] study the question of whether separately trained DNNs learn features that converge to similar spaces. One of their findings is that there are some features that are always learned by multiple networks while others are not. Wang et al. [133] study whether neural networks that have identical architectures but are trained using different initializations learn similar representations, and their conclusion is that the representations learned by the same layers of these network are not as similar as it is widely believed. Unlike all of these works, we seek to find DNN similarities from a functional perspective, and therefore, we only consider the ultimate behaviour of the model.

The work by Yellapragada [119] studies whether similarities in the representations learned by DNN models in their final layers (the layer after the last convolutional layer in their experiments) can be related to similarity in these models' functionalities. They study this by feeding meaningful canonical inputs to models, applying CKA to measure representational similarity, and applying Spearman rank correlation and Jensen-Shannon divergence to measure functional similarity. Their results show that there is not a strict correlation between these two.

2.3 Fuzz Testing Neural Networks

This work is also highly inspired by the work on generating inputs for fuzz testing DNNs. We briefly review this research area here.

DeepXplore [92] is a white-box differential testing approach to find incorrect behaviors among DNN models. The term differential here refers to relying on observing incosistencies in behavior of similar DNNs for finding erroneous behavior among them. It proposes neuron coverage as a test coverage metric and solves a joint optimization problem to maximize both neuron coverage and detecting the behavioral differences among similar models. Later research [42], however, finds neuron coverage to be suffering from problems such as generating unnatural inputs and being biased towards certain labels.

TensorFuzz [88] is a coverage-guided fuzzing technique for neural networks. Initialized by at least one valid seed, it mutates its list of seeds and adds the mutated point to the seeds list if new coverage is exercised, and to the test-cases if a user-defined property is satisfied. It measures coverage by looking at the activations that are activated (typically logits or one layer before): a new coverage is found if the distance between the vectors of activations is greater than a specified threshold and approximate nearest neighbor algorithm is used in this step to measure the distances.

DeepHunter [144] is a coverage guided fuzz testing framework for DNNs that proposes a metamorphic mutation strategy that needs domain knowledge to perform the mutation (in the paper, the strategy has been implemented for image recognition domain). It leverages a combination of paying attention to recently generated seeds as well as how many times a seed has been mutated for its seed selection. Their results show that coverage-guided fuzzing is more effective than random testing both in terms of improving the coverage and finding defects.

DiffChaser [145] is a black-box testing tool for DNNs focused on finding the disagreements among multiple models (mainly a model and its variants, for example, the compressed version of the model) to help in model debugging. They argue that the decision boundary of a model and its compressed version are mostly similar; thus, finding inputs that are near this boundary will help in finding the disagreements between these models.

Chapter 3

Code Clone Detection

In the first part of this chapter (Section 3.1), I discuss Oreo [108], a clone detection tool aimed at detecting a category of clones that lie in the spectrum of Type-3 to Type-4 clones, referred to as *Twilight Zone* clones. This is a joint work with Vaibhav Saini where Vaibhav was the lead author. I contributed to this work by doing the reproduction study experiments, assisting in developing the metrics calculator, preparing the training and testing datasets, collecting and running previous clone detectors and collecting their recall values using BigCloneEval, being a judge throughout the precision evaluation experiments, and packaging the Oreo artifact. A major part of the material covered in Section 3.1 is from the following paper, and is included here with the permission of the ACM:

- Saini, V., Farmahinifarahani, F., Lu, Y., Baldi, P., Lopes, C., "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), 2018, pp. 354-365. DOI: https://doi.org/10.1145/3236024.3236026

I then (Section 3.2) discuss a study of precision of clone detection tools [35], which highlights

the importance of studying these tools' precision by considering various clone types as well as the significance of Type III and beyond clones in precision studies. The material presented in Section 3.2 is from the following paper and is included here with the permission of IEEE.

-

3.1 Detecting Clones with Oreo

3.1.1 Introduction

Although there have been several clone detector proposed in the literature with different goals and following various approaches, there are very few ones that attempt at detecting code clones that have little or no syntactic similarity, but demonstrate functional (behavioral) similarity. Such clones start to appear at Type III and onwards (to Type IV). To better classify and recognize these clones, the popular clone benchmark BigCloneBench [125, 121] has specified subcategories between Type III and Type IV clones: Very Strongly Type III (syntactic similarity in the range of [90%, 100%)), Strongly Type III (syntactic similarity in the range of [70%, 90%)), Moderately Type III (syntactic similarity in the range of [50%, 70%)), and Weakly Type III which merges with Type IV (syntactic similarity in the range of [0%-50%)). As the similarity percentages show, starting from Very Strongly Type III towards Type IV, the syntactic similarity fades. A more detailed explanation of these subcategories can be found elsewhere [125, 121]).

To better illustrate clone types, Listing 3.1 shows one example method and Listing 3.2

Listing 3.1: Sequence Between Two Numbers: Original Method

```
1  // Original method
2  String sequence(int start, int stop) {
3      StringBuilder builder = new StringBuilder();
4      int i = start;
5      while (i <= stop) {
6          if (i > start) builder.append(',');
7          builder.append(i);
8          i++;
9      }
10     return builder.toString();
11 }
```

shows several clones of this method, from Type II to Weakly Type III. The original method takes two numbers and returns a comma-separated sequence of integers in between the two numbers, as a string. The Type II clone (starting from line #2 of Listing 3.2) is syntactically identical to the original method, 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 III clone (starting from line #14) has some lexical as well as syntactic differences, namely the use of a for-loop instead of a while-loop. Although harder than Type II, this subcategory of Type III is still relatively easy to detect. The moderate Type III clone (starting at line #24) differs even more from the original method: the name of the method is different (`seq` vs. `sequence`), the comma is stored in a local variable, and the type String is used instead of StringBuilder. This subcategory of Type III clones is much harder to detect than the previous ones. The weakly Type III clone (starting from line #35) differs from the original method by a combination of lexical and syntactic 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.

Clones that are moderately Type III and onward fall in the *Twilight Zone* of clone detection. Not many clone detectors are capable of operating on this zone, mainly due to the difficulty in detecting them. Oreo's goal is to improve the performance of clone detection for these hard-to-detect clones.

Listing 3.2: Sequence Between Two Numbers: Clones

```
// 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;
}
```

Oreo is a scalable method-level clone detector that is capable of detecting not just Type I through strong Type III 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 I to strong Type III 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.

There are two main insights based on which Oreo operates: (1) functionally similar pieces of code tend to do similar more fine-grained *actions*, which can be captured by the functions they call and the state they access; and (2) 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 the first part, Oreo uses a novel concept called *Action Filter* to filter out a large number of method pairs that don't seem to be doing the same fine-grained 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 (the second insight). To this aim, a machine learning model is trained based on a set of metrics derived from source code.

To curate the dataset of clone and non-clones, serving as the training dataset for the machine learning model, we used SourcererCC [111], a state of the art clone detector that has been shown to have fairly good precision and recall up to Type III clones (but not Type IV).

The contributions of this section are as follows:

(i) **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;

(ii) **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;

(iii) **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 used to train a model and predict clones in a scalable manner using the concepts introduced in this paper;

(iv) **Deep Neural network with Siamese architecture**. We propose Siamese architecture [14] to detect clone pairs. An important characteristic of this architecture is that it can handle the symmetry [85] 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. First in Section 3.1.2 I discuss a reproduction study that we did as a preliminary phase of building Oreo. Next, Section 3.1.3 discusses the Oreo clone detector and concepts integral to it, and Section 3.1.4 explains the deep neural network model used in Oreo. Section 3.1.5 elaborates on the evaluation of Oreo followed by a manual analysis of clone pairs in Section 3.1.6. Section 3.1.7 presents the limitations of this study, and finally, Section 3.1.8 discusses conclusions and future work.

3.1.2 Reproduction Study

In [114], Sheneamer and Kalita describe a metrics-based supervised machine learning approach to detect Type IV clones. We started the design of Oreo by reproducing a similar version of that work in order to verify whether, indeed, it was fruitful to train a model on metrics for purposes of clone detection. The results of our reproduction study were encouraging. Most importantly, this study provided us with valuable lessons as to the strengths

and pitfalls of this approach, which were critical to the development of Oreo.

The reproduction study was done on a sample dataset and consisted of three steps: *Feature Generation, Model Training, and Prediction*. Details of this process are explained throughout the rest of this section.

Dataset

For the purpose of this study, we used the dataset of 3,562 Java projects made available by Saini et al [109]. The dataset consists of 3,562 Java projects hosted on Maven [82]. The comprehensive list of projects with their version information is available at `http://mondego.ics.uci.edu/projects/clone-metrics/`.

Using this dataset we obtained the training dataset as described below:

1. *Clone-pairs*. The list of clone and non-clone pairs were collected by gathering the intra-project method-level clone detection results for the 3,562 Java 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*. In order to create the feature vector to be fed to the machine learning model for both training and inference, we calculated a set of method-level software metrics for each of the methods in this dataset. These metrics were calculated the using version 6 of the JHawk tool [50]. JHawk has been widely used in academic studies on Java metrics [41, 6, 20, 9, 10]. Table 3.1 shows the 25 metrics used for building the feature vectors. Many of these metrics are standard metrics. A set of metrics are derived from the Software Quality Observatory for Open Source Software (SQO-OSS) [112]. SQO-OSS is composed of well-established and validated software quality metrics, which can be computed either from source code or from sur-

rounding community data. More details on these metrics and the dataset can be found elsewhere [109].

Experiments

The experiments carried out to implement the whole process of our idea are as follows.

- **Feature Preparation.** In order to leverage the method-level software metrics described above and calculated by JHawk in to train a classifier that can distinguish clones and non-clones, a new dataset needed to be created such that these features can capture the relationship between the two methods in each method pair and how they differ. To this aim, we calculated the percentage that two methods differ in terms of each metric in a method pair. Having a clone pair with two methods named *M1* and *M2*, if metric f calculated for *M1* is *f1* and the same metric calculated for *M2* is *f2*, the percentage difference between *f1* and *f2* is calculated by the formula in 3.1. The set of 25 features created for a method pair based on this forms a *Feature Vector*.

$$Percentage - diff(f1, f2) = \frac{|f1 - f2|}{Max(f1, f2)} * 100 \tag{3.1}$$

- **Training and Testing Data Creation** We separated the whole dataset at hand randomly to two parts: 60% for *training*, and the rest for *testing*. For the training dataset, each Feature Vector was accompanied with its corresponding *is_clone* label calculated using SourcererCC. To have a manageable sized dataset, we filtered out pairs for which the percentage difference in NOS was more than 30%.

- **Model Selection** To select a target model, we performed 10-fold cross validation, (a common form of N-fold cross validation [99]), with various algorithms. To speed-up the process, we did the training using three different samples taken from our dataset (which

Table 3.1: 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 effort 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

was 45GB in size): 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 generic and independent of characteristics of a single dataset sample. Each of the three sampled datasets were split into 70% training and 30% testing sets.

We experimented with various classification algorithms: K-Nearest Neighbors (KNN) [142], Naive Bayes [142] , Classification and Regression Trees (CART) [142], Support Vector Machine (SVM) [142], Logistic Regression (LR) [31], Linear Discriminant Analysis (LDA) [49], Random Forest [127], and AdaBoost [142]. For models trained using each algorithm, we measured their relative precision and recall with respect to SourcererCC , and the time each of them took to complete the experiments (in seconds). Table 3.2 shows the results of 10-fold cross validation experiments with a model trained using each algorithm. Here, T shows the total time spent on 10-fold cross validation. In addition, Table 3.3 shows the results of training and testing the same algorithms on the training and the hold-out testing datasets. Here, we show the time to train models (T_T) and the time taken to perform the prediction using each model (T_P).

As the tables show, overall, the results were promising showing that it is possible to detect clone pairs using the selected software metrics. Although all algorithms showed promising results, Random Forest stood out both in terms of accuracy and the time it took to perform the training and the inference (prediction). Therefore, we further experimented with the Random Forest algorithm using different configurations of this algorithm using on 100-K rows dataset. The results of these experiments are shown in Table 3.4. As this table shows, the configurations on the first two rows show the best results while spending the less amount of time. We chose the configuration on the first row as our final selected model as it needed the least amount of time for both training and prediction.

Table 3.2: Results of 10-fold Cross Validation on the Sampled Rows of the Training Dataset

Algorithm	10 Thousand Rows				50 Thousand Rows				100 Thousand Rows			
	Prec	Rec	F1	T	Prec.	Rec	F1	T	Prec	Rec	F1	T
KNN K=5	91	94	93	1.5	92	97	94	30.3	93.4	97	95	135.3
KNN K=10	92	92	92	1.6	92	96	94	33.6	93	97	95	145.3
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
Rand Forest	93	94	94	0.5	94	96	95	2.7	95	97	96	5.7
AdaBoost	93	934	94	2.9	94	96	95	11.7	95	97	96	24.4

Table 3.3: Results of Models Training and Prediction on the Hold-out Testing Part of the Sampled Rows of Dataset

Algorithm	10 Thousand Rows					50 Thousand Rows					100 Thousand Rows				
	Prec	Rec	F1	T_T	T_P	Prec	Rec	F1	T_T	T_P	Prec	Rec	F1	T_T	T_P
KNN K=5	91	94	93	0.02	0.71	93	98	95	0.23	17	94	97	96	0.82	58
KNN K=10	90	92	91	0.03	0.67	93	97	95	0.23	21	94	97	95	0.7	69
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	0.01	92	97	94	0.32	0.03	93	96	95	0.71	0.05
SVM	93	89	91	3.6	1.4	97	93	95	211	35	96	94	95	2079	143
LR	89	93	91	0.13	0.01	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
Rand 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	2.92	0.09

Table 3.4: Different Configurations of Random Forest Classifier on 100 Thousand Rows Sample

#Est.	Max Depth	Min Samp. Split	Min Samp. Leaf	Max Features	10- Fold Cross Validation				30% Hold-out Test Set				
					Prec	Rec	F1	T	Prec	Rec	F1	T_T	T_P
5	20	10	5	$\sqrt{\#Feat.}$	95	96	95	5.6	94	96	95	0.43	0.06
5	Default	Default	Default	Default	95	97	96	5.7	95	96	95	0.49	0.09
10	10	20	5	$\sqrt{\#Feat.}$	94	95	95	8.1	95	95	95	0.82	0.084
50	5	Default	Default	$\sqrt{\#Feat.}$	95	86	90	26.1	95	85	90	2.81	0.18
25	10	Default	Default	$\sqrt{\#Feat.}$	95	96	96	17.5	95	95	95	1.83	0.126

- **Evaluation.** To measure the performance of the selected model on large-scale and unseen data, we trained a final model on the whole training dataset. Since this dataset had an unbalanced distribution of clones and non-clones (there were approximately 7 times more clone pairs compared to non-clone pairs), we performed sampling over this data to have a balanced distribution of labels. Training the model on 60% training data and testing it using the 40% unseen testing data, resulted in 88% Precision and 97% Recall with respect to SourcererCC's results. Model training took 570 seconds and prediction took 345 seconds over this dataset.

Lessons Learned

This reproduction study provided us with insights that an approach based on a combination of software metrics and machine learning is effective in detection clone pairs. It also helped us detect the main issues that need to be tackled when implementing a tool using this approach. A summary of the lessons learned include:

- Software metrics have a high potential for being used in the context of code clone detection. They, however, need to be engineered so that they can capture meaningful properties pertaining to the pair of methods being analyzed. A manual inspection of false negative pairs in this study revealed that when two features have a small value,

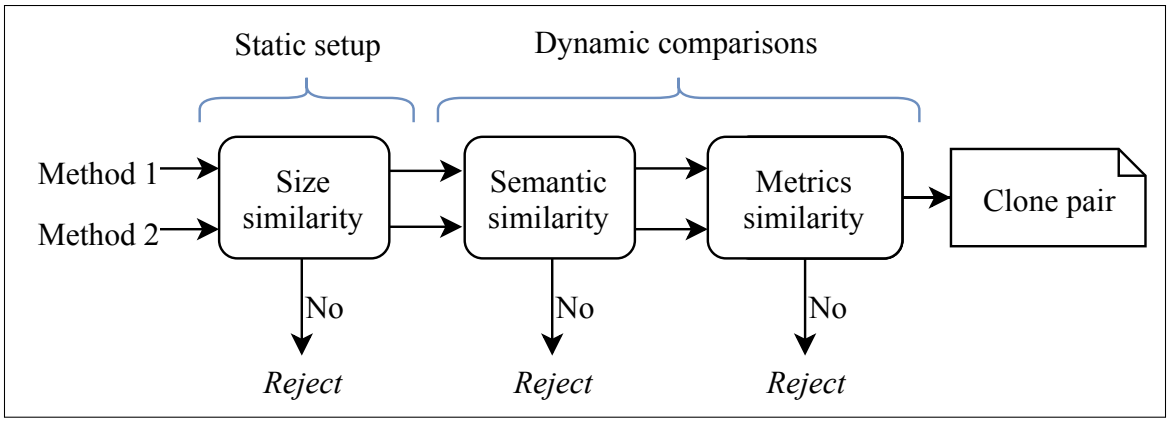


Figure 3.1: Overview of Oreo.

even with a very small difference, their Percentage Difference becomes large. Consider 1 and 2 which have 50% percentage difference; this value equals to the percentage difference between 10 and 20, but these two cases should be treated differently. It is important to pay attention to such cases.

- Random Forest algorithm has a better performance compared to other algorithms for this problem using the analyzed metrics.

- A manual analysis of false positives revealed many code fragments that are not actual clone pairs, but have identical structures. Hence, we need additional mechanisms that can recognize semantic similarity beside the structural similarity, .

- This approach is susceptible to the problem of *Candidate Explosion* as every possible method pair is being generated ($O(n^2)$). In this study, the problem was tackled by limiting the number of clone pairs based on their differences in terms of number of statements. However, a more robust approach is needed in order for this approach to be able to scale to larger datasets.

3.1.3 The Oreo Clone Detector

Oreo builds upon the lessons learned from presented in the previous chapter. It utilizes software metrics learned by a machine learning component to detect clones with low syntactic but high semantic similarity (twilight zone clones). In order to overcome two problems of the existence of several false positives and candidate explosion, faced through the reproduction study, we add a semantic signature concept that works based on the fine-grained actions performed by methods. The concept of semantic signature helps in making Oreo's approach scalable to large repositories. This is accompanied by a simple size-based heuristic that eliminates a large number of unlikely clone pairs. Using these two, Oreo becomes able to process very large datasets consisting of hundreds of millions of methods. Figure 3.1 gives an overview of Oreo.

Size Similarity Sharding

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 IV clones. However, in all our experiments, we observed little to no impact on the recall of other clone types.

This is implemented by counting the number of language tokens present in two methods and

do the filtering as follows: 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 $x \times T \leq y \leq \frac{x}{T}$.

Semantic Similarity: The Action Filter

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.

As an example of *Action tokens*, consider the code in Listing 3.3, 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] These *Action tokens*, more than the identifiers chosen by the developer, or the types used, are a 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. 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 $freq$ is the number of times this token appears in the method.

In order to speed up comparisons, we create an inverted index of all the methods in a given

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

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.

Using the notion of method calls to find code similarity has been previously explored by Goffi et al. [39], 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 III clones [111].

Listing 3.3: Action Filter Example

```
1  public static String getEncryptedPassword(String password) throws InfoException {
2      StringBuffer buffer = new StringBuffer();
3      try {
4          byte[] encrypt = password.getBytes("UTF-8");
5          MessageDigest md = MessageDigest.getInstance("SHA");
6          md.update(encrypt);
7          byte[] hashedPasswd = md.digest();
8          for (int i = 0; i < hashedPasswd.length; i++) {
9              buffer.append(Byte.toString(hashedPasswd[i]));
10         }
11     } catch (Exception e) {
12         throw new InfoException(LanguageTraslator.traslate("474"), e);
13     }
14     return buffer.toString();
15 }
```

Metrics Similarity

Method pairs that survive the size filter and the *Action filter* are analyzed for their similarity based on a set of software metrics. This is done using a machine learning classifier whose feature vectors are these metrics calculated for both methods. The intuition for using metrics as the final comparison after size and actions similarity 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.

The set of software metrics used here are a subset of metrics used during the reproduction study plus a set of metrics added after analyzing the pairs predicted by the final model of reproduction study. Table 3.5 shows the final 24 method level metrics used. The decision of which of the software metrics from the reproduction study to include was 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.

The metrics added to the metrics from the reproduction study are marked with $*$ in the table. They are aimed at capturing the information on how many times various types of literal are used by a method and were derived after noticing that there many Twilight Zone 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.

Table 3.5: Final Method-Level Software Metrics Used in Oreo

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

3.1.4 Learning Metrics

In this section, we describe the dataset used to train the machine learning model, and also, the trained model itself and its selection process.

Dataset Curation

To prepare the dataset, we downloaded 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 [122]. To get *isClone* labels, similar to the reproduction study, we used SourcererCC. From this dataset, we randomly sampled 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.

Deep Learning Model

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. This eliminates the need for features engineering to compute the relationship between two methods in terms of the 24 software metrics. The relationships are automatically derived throughout the training process, therefore tackling another issue that we encountered during the reproduction study.

Here we use a Siamese architecture neural network [14] 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 [14]. Another important characteristic of this architecture is that it can handle the symmetry [85] of its input vector. Which means, presenting the pair $(m1, m2)$ to the model will be the same as presenting the pair $(m2, m1)$. This ability is achieved by applying the same operation to each component of the pair by using two identical sub neural networks. This is crucial in clone detection, the equality of clone pair $(m1, m2)$ with $(m2, m1)$ is an issue that should be addressed while detecting or reporting clone pairs. 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.

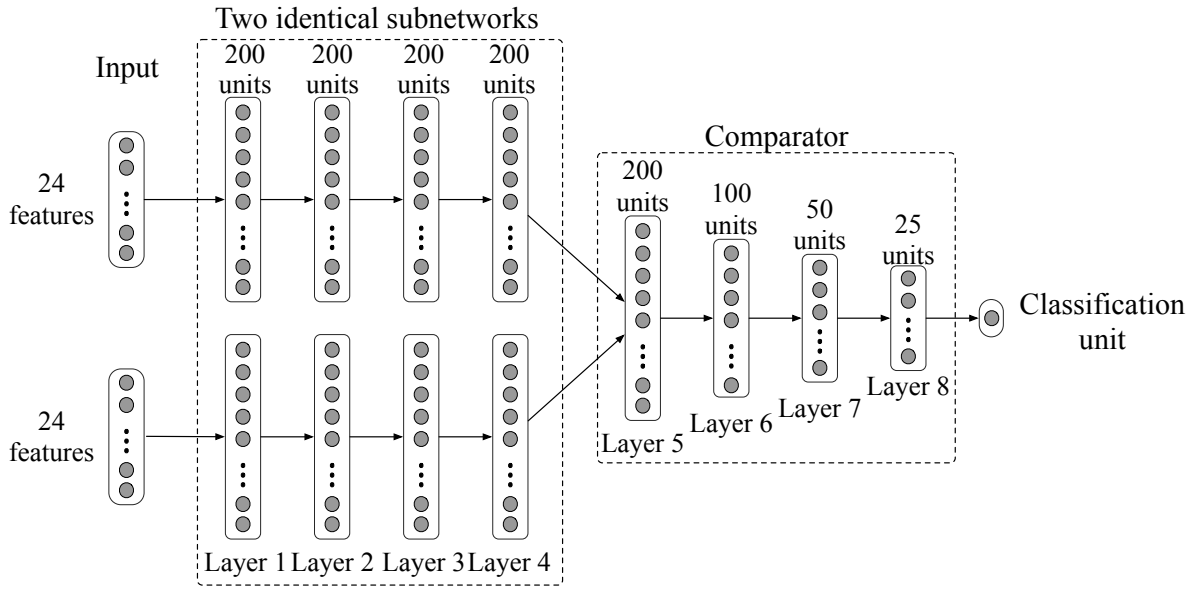


Figure 3.2: The Trained Siamese Architecture Model

Figure 3.2 shows the Siamese architecture model trained for Oreo. Here, the input to the model is a 48 dimensional vector created using the selected 24 metrics described.This 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.The outputs of the two subnetworks are then concatenated 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 the 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. A value above 0.5 is interpreted as a clone pair has been detected.

In this model, to prevent overfitting, dropout regularization technique [15] is applied to every other layer. In our experiment, we achieved the best performance with 20% dropout. The

loss function is the relative entropy [75] between the distributions of the predicted output values and the binary target values for each training example. Training was carried out by stochastic gradient descent with the learning rate of 0.0001. The learning rate is reduced by 3% after each epoch to improve the convergence of learning. The parameters are initialized randomly using 'he normal'[44], 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.

3.1.5 Evaluation of Oreo

We compare Oreo's detection performance against the latest versions of the four publicly available clone detection tools: SourcererCC [111], NiCad [104], CloneWorks [126], and Deckard [51].

We also wanted to include tools such as SeByte [64], Kamino [87], JSCTracker [34], Agec [59], and approaches presented in [140, 138, 114, 129], which claim to detect Type IV 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 [37] and [52] said that they do not have implementations for detecting Java clones (They work either on C or C++ clones).

As Type I and Type II clones are relatively easy to detect, we focused primarily on Type III clone detectors. The configurations of these tools, shown in Table 3.6, are based on our discussions with their developers, and also the configurations suggested in [122]. 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. In the table, *MIT* stands for minimum tokens, Θ stands for similarity threshold (for NiCad, it is difference threshold, and for Oreo it is *Action filter* threshold), Γ stands

for threshold for input partition used in Oreo, and *BIN* and *IA*, respectively stand for blind identifier normalization and literal abstraction used in NiCad.

Recall

The recall of these tools is measured using Big-CloneEval [124], which performs clone detection tool evaluation experiments using BigCloneBench [122], a benchmark of real clones. Big-CloneEval reports recall numbers for Type I (T1), Type II (T2), Type III, and Type IV 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 [18, 122].

To report numbers for Type III and Type IV 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 III (VST3), where the similarity is between 90-100%, ii) Strongly Type III (ST3), where the similarity is between 70-90%, iii) Moderately Type III (MT3), where the similarity is between 50-70%, and iv) Weakly Type III/Type IV (WT3/4), where the similarity is between 0-50%. Syntactical similarity is measured by line and by language token after Type I and Type II normalizations.

Table 3.6 summarizes the recall number for all tools. The recall numbers are summarized per clone category. The numbers in the parenthesis after each category title 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

Table 3.6: Recall and Precision Measurements on BigCloneBench

Tool	Recall												Prec.	Tool config.
	T1 (35,802)		T2 (4,577)		VST3 (4,156)		ST3 (15,031)		MT3 (80,023)		WT3/T4 (7,804,868)			
	%	#	%	#	%	#	%	#	%	#	%	#		
Oreo	**100**	**35,798**	**99**	**4,547**	**100**	**4,139**	89	13,391	**30**	**23,834**	0.7	57,273	89.5	MIT=15, $\Theta = 55\%$,$\Gamma = 60\%$
SourcererCC	100	35,797	97	4,462	93	3,871	60	9,099	5	4,187	0	2,005	97.8	MIT=1, $\Theta = 70\%$
CloneWorks	100	35,777	99	4,544	98	4,090	**93**	**13,976**	3	2,700	0	35	98.7	MIT=1, $\Theta = 70\%$,Mode=Agg.
Nicad	100	35,769	99	4,541	98	4,091	93	13,910	0.8	671	0	12	**99**	MIL=6,BIN=True, IA=True,$\Theta = 30\%$
Deckard	60	21,481	58	2,655	62	2,577	31	4,660	12	9,603	**1**	**780,487**	34.8	MIT=50, Stride=2,$\Theta = 85\%$

Deckard are taken from SourcererCC's paper [111], 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 3.6 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 (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 iden-

tifiers 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.

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 [111].

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 two judges, who are also the authors of this work. 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 I, Type II, Type III, and Type IV clones. Out of the 2,000 pairs manually inspected, both judges gave the same judgment for 1,846 pairs making the inter rater agreement as 92.3%. The conflicts in the judgments were then resolved by discussions, which always ended up in consensus simply by invoking the definitions.

Table 3.6 shows precision results for all tools. We found that the precision of Oreo is 89.5%. All other tools except Deckard performed better than Oreo. Deckard's precision is the lowest at 34.8% and Nicad's precision is the highest at 99%. 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.

The recall and precision experiments demonstrate that Oreo is an accurate clone detector capable of detecting clones in Type I, Type II, Type III and in the Twilight Zone. Also, note that Oreo is trained using the clone pairs produced by SourcererCC. As SourcererCC does not perform well on harder-to-detect categories like ST3, MT3, and WT3/T4, our current training dataset lacked such examples. To address this issue, in future we will train Oreo with an ensemble of state of the art clone detectors.

3.1.6 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.

Here, we present two examples of clone pairs with high semantic similarity and low syntactic similarity. Listing 3.4 shows one of the classical examples of Type IV 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* 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 IV clone here can be attributed to chance. Nevertheless, these two implementations share enough semantic and structural similarities to be classified as a clone

Listing 3.4: Clone Pair Example: 1

```
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 3.5: Clone Pair Example: 2

```
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 3.6: False Positive Example

```
1  public static String getHexString(byte[] bytes) {
2      if (bytes == null) return null;
3      StringBuilder hex = new StringBuilder(2 * bytes.length);
4      for (byte b : bytes) {
5          hex.append(HEX_CHARS[(b & 0xF0) >> 4]).append(HEX_CHARS[(b & 0x0F)]);
6      }
7      return hex.toString();
8  }
9  ----------------------------------------
10 String sequenceUsingFor(int start, int stop) {
11     StringBuilder builder = new StringBuilder();
12     for (int i = start; i <= stop; i++) {
13         if (i > start) builder.append(',');
14         builder.append(i);
15     }
16     return builder.toString();
17 }
```

pair by Oreo.

Another example is illustrated in Listing 3.5 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 the character '/' in its input string (line 9). We were unsure whether to classify this example as a WT3/T4 or a 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 3.6. *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.

3.1.7 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 [136]. We mitigated 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 mitigated it by involving two judges. This bias, however, can be further reduced by involving more judges.

3.1.8 Conclusions and Future Work

In this section, I discussed 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 an 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. We compared Oreo with four other state of the art tools on a standard benchmark and demonstrated 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.

3.2 On Precision of Clone Detectors

3.2.1 Introduction

While code clone detection techniques for traditional use cases have been documented in the literature since the early 90s [148, 56, 29], the recent advent of freely-accessible massive source code repositories created opportunities for new use cases, including mining library candidates [48], license violation detection [8], code refactoring [139], code quality analysis [96, 105], detecting similar mobile applications [27], aspect mining [24], and analyzing programmers' behavior [147]. These applications further motivate researchers to devise novel clone detection tools and techniques. These tools have not only enabled the practical discovery and management of clones, but also assisted researchers in conducting studies to gain insight into the practice of code cloning.

A survey in 2013 noted the presence of more than 70 clone detection tools and techniques in the literature [98]. More have been developed since then. While these tools exist, our knowledge of their correctness is fairly limited, as their evaluations tend to be done on diverse datasets. There are some efforts to simplify evaluating the performance of clone detection tools [19], [120, 121], [73]. As the field advances and the number of tools grow, the need to develop benchmarks and frameworks for assessing these tools is growing as well.

There are three main dimensions to assess clone detection tools and techniques: how good the classification is (correctness), the tool's scalability, and its execution time [110]. Of these, correctness is the most important and is usually measured by two important metrics: *Precision* and *Recall*. Precision is the percentage of reported clones that are true clones; recall is the percentage of true clones in the corpus that are identified as such. To estimate recall of a clone detection tool, the tool is generally run on a dataset with a set of tagged clone pairs, and then, recall is estimated by measuring the fraction of true clone pairs retrieved.

The popular BigCloneEval [125] facilitates the measurement of recall by providing a curated dataset of manually tagged clone pairs. However, BigCloneEval, and the methodology used to produce it, cannot be followed for measuring precision, because not all possible clone pairs are tagged, a required condition to measure precision. In fact, evaluating precision of clone detection tools is particularly challenging because it requires each of the candidate clone pairs retrieved by a tool to be manually labeled as a true clone or not. This task is largely manual, extremely time consuming, and often requires expertise in reviewing clone candidates. Therefore, precision of clone detection tools is often not reported. When it is reported, it is estimated by manually inspecting a statistically significant random sample of clone pairs reported by the tool [111, 108, 134].

Goals and Contributions of This Work

The established practice in the literature of sampling clone pairs for manual inspection is agnostic to clone type: the estimated precision does not differentiate among different types of clones. However, different clone detectors focus on different types of clones: some target only easy-to-detect clones, while others attempt to use more sophisticated notions of similarity, therefore broadening the scope of what they are looking for. Two clone detectors can have the same precision while detecting very different types of clones. This means that the measured precision of tools may not be directly comparable without further considerations. Comparative results can be made more fair and enlightening by including clone type information in precision measurements, and/or ignoring types I and II entirely.

This chapter presents the result of systematic experiments to measure the precision of eight code clone detection tools. These experiments were conducted in two parts: the first set of experiments was done to measure the undifferentiated precision of each tool, which is the established practice in the literature; the second set of experiments measured each tool's precision separately for different types of clones. A challenge in measuring precision per

clone type is that clone detectors do not include type information when reporting clone pairs. In order to overcome this, we used algorithms, based on clone types definitions, to identify different clone types. Through the experiments, this work attempts to raise appreciation and motivation for measuring and reporting type-based precision in conjunction with undifferentiated precision for improving the evaluation of clone detection tools in the future.

The key contributions of this study are as follows: (i) Benchmarking undifferentiated and type-based precision measurements for eight clone detection tools; our work shows that the detection of types I and II is essentially a solved problem, as several tools have almost perfect precision and recall on these types; (ii) Experiment-driven analysis revealing the importance of type-based precision evaluation for comparing clone detection tools; specifically, given the previous point, we show that it is important that the tools report on their precision for Type III and beyond.

In the experiments reported here, three judges spent a total of 115 person hours to independently review 12,800 candidate clone pairs (i.e. the output of eight code clone detection tools) and manually classify them as true or false positives. We encourage researchers to reproduce, replicate, and reuse the dataset constructed in this study. To this end, we have made publicly available the dataset of manually tagged clone pairs. The dataset can be downloaded from http://mondego.ics.uci.edu/projects/precision-study/.

The remainder of this chapter is structured as follows. In Section 3.2.2, we explain the design of our study and the components of it. Section 3.2.3 presents the undifferentiated experiments, and Section 3.2.4 elaborates on the type-based experiments. In Section 3.2.5 we discuss about some qualitative aspects of the experiments. Related work is described in Section 3.2.6, and the threats to validity are presented in Section 3.2.7. Finally, conclusions and future work are discussed in Section 3.2.8.

3.2.2 Study Design

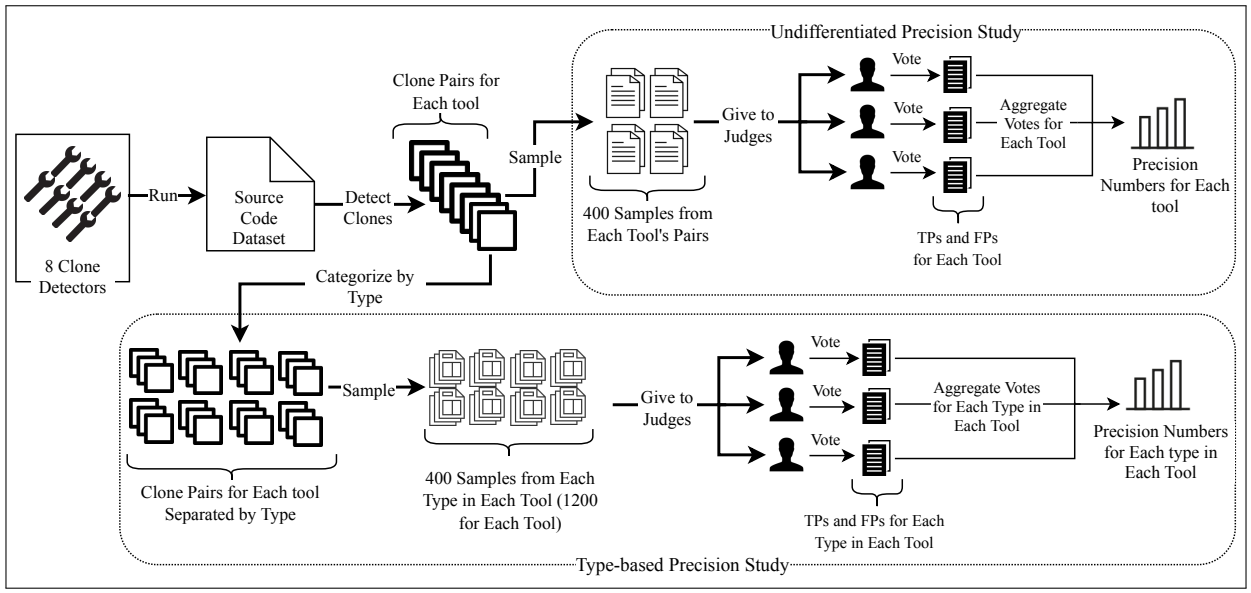


Figure 3.3: Overview of the Precision Study

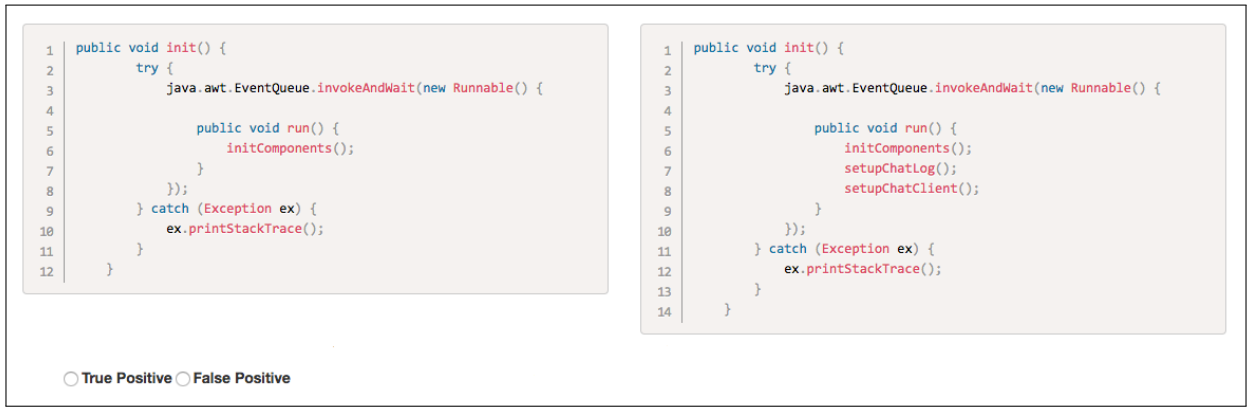


Figure 3.4: Voting on a Clone Pair

Figure 3.3 shows an overview of the process we followed to carry out the precision experiments. The process starts with running eight clone detection tools on the source code dataset, and detecting clones with each tool. After having the clone pairs detected by each tool, we conduct the two sets of precision experiments. In the undifferentiated study, illustrated at the upper part of the figure, first, for each tool, a random sample of 400 pairs is selected from its clone pairs. Then, these samples are presented to three judges, and each judge, independently, goes through all pairs, tagging them as either True Positive (TP) or

False Positive (FP). To facilitate code comprehension and the voting process, judges were presented with code fragments in a side by side manner as shown in Figure 3.4. Finally, votes for each tool are aggregated, and precision numbers for the eight tools are calculated. Aggregation of votes can be done in a number of ways. One of them is by taking the majority vote, which means that a clone pair is declared as true positive when at least two of judges vote as such. Precision numbers reported in Section 3.2.3 and Section 3.2.4 are calculated using this approach. We revisit and discuss this approach in Section 3.2.5.

To conduct the type-based experiments, after getting the tools' clone pairs, we follow the process shown on the bottom part of Figure 3.3. First, we classify the clone pairs into three clone types (Type I, Type II, and Type III). Then, from the clone pairs in each set of clone types for each tool, a sample of 400 pairs is made, and presented to three judges. Hence, each judge is presented with 1200 pairs for each tool. Finally, votes are recorded and aggregated to estimate the precision of each tool in each of the clone types.

Through the rest of this section, we first introduce the clone detection tools that we used in this study, and then, we describe the dataset used to run the tools on.

Clone Detection Tools

The set of tools targeted by this study are clone detectors that are capable of detecting Type III clones. To select these tools, we followed the following process: (i) we looked for the tools that are used by researchers in recent studies to compare different clone detectors: NiCad [104], SourcererCC [111], SimCad [131], iClones [38], CPD [1], CCFinder [62], CloneWorks [126], Deckard [51]; (ii) we searched for the tools that are recently proposed: Oreo [108], CCAligner [134]; (iii) we also looked for the tools that dive deeper into Type III category: SeByte [65], Kamino [87], Agec [60], JSCTracker [34], EqMiner [53], and CCCD [74]; (iv) we tried to get a stable version of these tools. We contacted the au-

Table 3.7: Clone Detection Tools' Characteristics and Configurations

Tool Name	Year	Approach	Clone Type	Configuration
NiCad	2008	Text based	1,2,3	Min lines=6, Blind identifier normalization =True, Literal abstraction=True, Difference threshold=30%
SourcererCC	2016	Token and Index based	1,2,3	Min tokens=1, Similarity threshold= 70%
SimCad	2013	Simhash based on Fingerprinting	1,2,3	Min lines=6, Greedy transformation=True, Unicode support=True
iClones	2009	Suffix Trees based	1,2,3	Min tokens=50, Min block=20
CloneWorks (A)	2017	Token and Index based	1,2,3	Min tokens=1, Similarity threshold = 70%, Mode= Aggressive
CloneWorks (C)	2017	Token and Index based	1,2,3	Min tokens=1, Similarity threshold = 70%, Mode= Conservative
Oreo	2018	Metrics and Machine learning based	1,2,3	Min tokens=15, Action filter threshold= 55%, Input partition threshold= 60%
CCAligner	2018	Token based	1,2,3	Min lines=6, Similarity threshold = 60%, Edit distance=1, Window size=6

thors of the tools for which we could not find an executable version. In correspondence with some of these tools' authors we either did not receive any response, or were told that they do not have a packaged tool available, or were informed that their tool does not work on Java (the language of our dataset), or that they no longer are maintaining the tool. (v) Finally, we discarded any tool for which we could not find an executable version or any tool that was not capable of being run on Java. At this point, we were left with nine tools: Nicad, SourcererCC, SimCad, iClones, CPD, CloneWorks, Deckard, CCAligner, and Oreo. We excluded Deckard and CPD, because they both report clones in clusters, and to generate clone pairs, every code fragment needs to be pared with others in the cluster. This process

produced more than 175GB of data (for each tool) that was hard to manage.

After following the process, we were left with seven tools. One of the tools (CloneWorks) comes in two different modes (aggressive (A) for higher recall and conservative (C) for higher precision), and we used both modes. Hence, in the final list, we had eight tools. These tools are described in Table 3.7. The first column of this table shows the tool name, the second column denotes the year the tool was proposed, and the third column presents the overall approach followed by the tool to detect clones. The fourth column shows what types of clones the tool tries to find; the information for this column was either obtained by referring to tools' papers, or by consulting with Svajlenko et al. study [121]. Finally, the last column shows the configurations used to run each tool. These configurations were obtained by consulting the tools' authors, or by referring to the configurations reported in their corresponding papers, or the ones reported by Svajlenko et al. [121].

Dataset

The dataset of this study is the dataset curated for BigCloneEval, a tool for conducting recall studies on clone detectors [125]. This dataset has been developed using IJaDataset-2.0 [7]. IJaDataset-2.0 is a large Java repository consisting of 2.3 million Java source files (365MLOC) from 25,000 open-source software projects [125]. Each of the target tools of this study has been run on this dataset, and precision analyses have been conducted on their reported clone pairs.

3.2.3 Undifferentiated Precision Experiments

This section describes the results for precision of the tools, ignoring the existence of different types of clones.

Experiment Setup. We randomly sampled 400 method-level pairs from the clone pairs reported by each tool, a statistically significant sample size used in precision studies [134, 111, 108]. We ensured that the methods in each pair have greater than or equal to 50 language tokens which is a standard size in clone studies [111, 108]; to this aim, we used the number of language tokens provided by BigCloneBench [121]. Then, three judges independently went through each sample, and marked the pairs in these samples as either false or true positive. Judges were kept blind from the source (the tool that generated the clone pairs) of each sample to avoid any bias in judgments. Judges were also asked to keep track of time that they spent on experiments. Together, they spent around 35 person hours to complete these experiments. We note that all three judges are experts in software clones and are aware of clone and clone types definitions.

Results. The precision results are presented in the second column of Table 3.8. As we see from the table, iClones has a perfect precision score; SourcererCC, CloneWorks (C), and NiCad show very high precision ($> 90\%$); Oreo, CloneWorks (A), and CCAligner show a decent precision ($> 70\%$), whereas SimCad at 5.5% scored very poorly.

These experiments, however, do not give us information on how each tool performs in different clone types. Does iClones perform perfectly in detecting all clone types? Does SimCad have a poor precision in all clone types? One may argue that these undifferentiated numbers, if studied with differentiated (type-based) recall numbers, can give such information. However, while these two metrics together give a good idea about the performance of tools, they still do not convey on which category a given tool did or did not perform well in terms of precision.

To elaborate on this, we present the recall results for these tools in Table 3.8 as well (columns under the Recall header), and discuss one example case. Recall numbers are calculated using BigCloneEval [125], a tool that estimates clone detectors' recall for various clone categories using BigCloneBench.

Table 3.8: Undifferentiated Precision and Differentiated Recall

Tool Name	Precision	Recall					
	(400 Pairs)	Type I	Type II	VST3	ST3	MT3	WT3/T4
NiCad	94%	100%	99%	98%	93%	0%	0%
SourcererCC	98.5%	100%	97%	93%	60%	5%	0%
SimCad	5.5%	100%	98%	91%	48%	7%	0%
iClones	100%	100%	77%	34%	9%	0%	0%
CloneWorks (A)	77.25%	100%	99%	98%	93%	3%	0%
CloneWorks (C)	95.75%	100%	97%	92%	60%	5%	0%
Oreo	82.5%	100%	99%	100%	89%	30%	0%
CCAligner	71.25%	100%	99%	97%	70%	10%	0%

By looking at Table 3.8, we observe that the recall numbers for SourcererCC and SimCad are similar for most clone categories, but the undifferentiated precision numbers for these two tools are very different. With these two sets of numbers, we cannot tell on which clone categories SimCad's precision is worse than SourcererCC's. A type-based precision analysis is required to show where these two tools are different in terms of precision.

3.2.4 Type-based Precision Experiments

Table 3.9: Number of Clone Pairs Reported by Each Tool Per Type

Tool name	Total	Total Parsed	Type I		Type II		Type III	
			#	%	#	%	#	%
NiCad	7,144,918	7,138,729	1,516,232	21%	317,371	5%	5,305,126	74%
SourcererCC	15,689,823	15,689,823	1,835,510	12%	8,317,571	53%	5,536,742	35%
SimCad	33,720,051	33,650,471	808,226	2%	235,480	1%	32,606,765	97%
iClones	6,991,429	3,951,090	3,601,213	91%	131,920	3%	217,957	6%
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%
Oreo	4,186,474	4,186,474	942,078	22%	198,691	5%	3,045,705	73%
CCAligner	4,253,798	4,252,328	938,421	22%	178,766	4%	3,135,141	74%

The results presented earlier in Section 3.2.3 can be used to give an overall view on the

precision of the tools, however, they miss out on one important information: If a tool's undifferentiated precision is low/high, does it mean that its precision is low/high on all types of clones? Also, different tools have different detection capabilities across clone types that can affect the undifferentiated precision numbers. For example, consider a case where a tool detects a large number of Type I and Type II clones and a significantly fewer number of Type III clones. A statistically significant sample will therefore contain more instances of Type I and Type II clone pairs in comparison to the Type III clones. Since inherently, it is relatively less error prone to detect Type I and Type II clone pairs, the undifferentiated precision of this tool will be high. Now consider an example of a tool that detects significantly larger number of Type III clones than Type I and Type II clones. A statistically significant sample of its output, when analyzed, will have more number of Type III clones. Since detecting Type III clone pairs is more complicated and error prone than detecting Type I and Type II clones, the undifferentiated precision of this tool may be low. However, it is possible that this tool has near perfect precision for Type I and Type II clone pairs; an information which is missing in the results of undifferentiated precision experiments. To study these concerns, we classify the clone pairs reported by each tool into Type I, Type II, and Type III clone types. The rest of this section elaborates on how we did this classification.

Table 3.9 presents the results of this classification exercise. The first column of this table shows the 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 (we explain the reason for parsing the clone pairs in the rest of this section). 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 in that type. We also present Figure 3.5 to aid the visualization of these numbers.

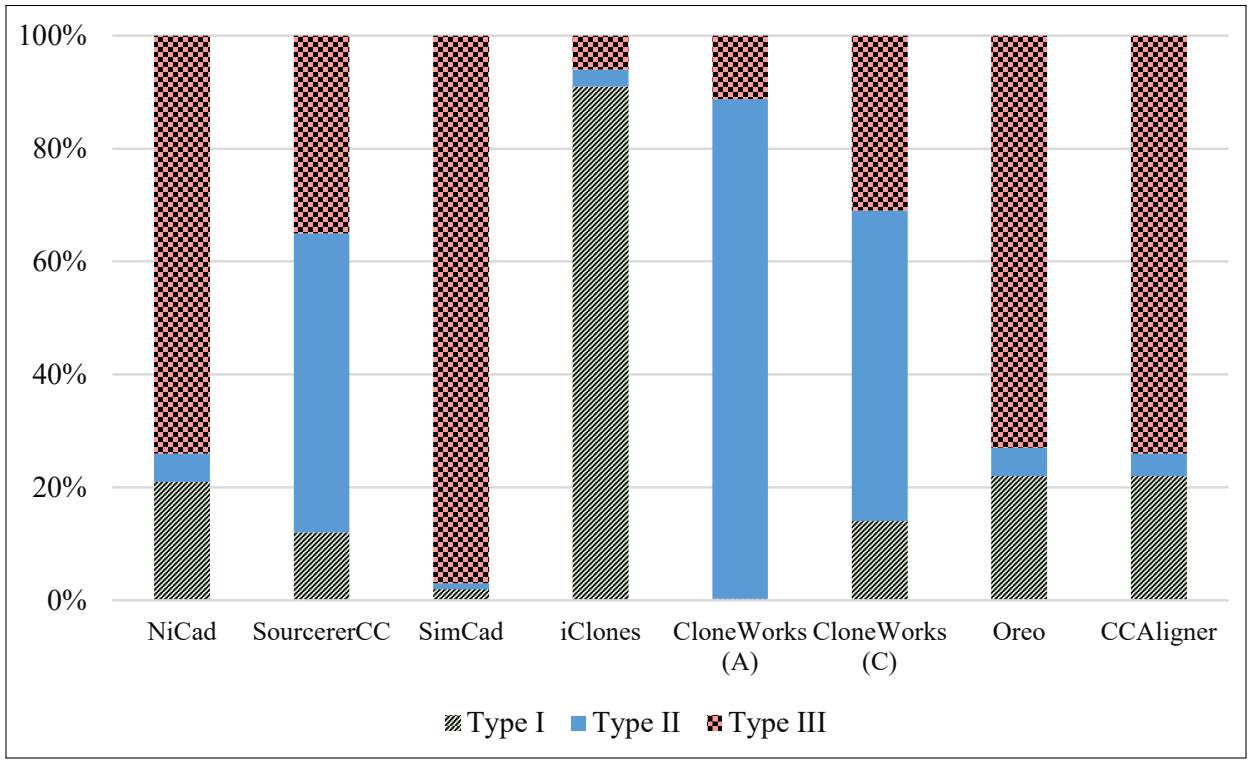


Figure 3.5: Distribution of Tools' Clone Pairs Per Clone Type

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, most of the clone pairs reported by iClones are of Type I (91% of all its reported pairs), and for other tools, the set of reported Type I pairs is less than 25%. 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 SimCad is the greatest among others (97%), and then the largest numbers are for NiCad and CCAligner being 74%, and Oreo being 73%.

This classification gives insight into the detection behavior of different tools, and also, on what to expect from the undifferentiated precision experiment of each of these tools. For example, 97% of the total clone pairs detected by SimCad are categorized into Type III. It is

natural that a random sample of SimCad's clone pairs will contain a large number of Type III pairs, and the precision experiment conducted on this sample will more or less tell us about the performance of SimCad on Type III clone category. Hence, the undifferentiated experiment alone is not sufficient to understand the performance of SimCad on Type I and Type II categories.

In the rest of this section, we first introduce the methods that we developed to classify tools' clone pairs into types, and then, present the results of type-based analysis and discuss how these results are different from undifferentiated results.

Classification of Clone Pairs

One of the reasons why doing type-based precision studies is a hard task, is that most tools do not report clone type information along with their clone pairs. Hence, there needs to be mechanisms for classifying the reported clone pairs per type. We operationalized algorithms to classify the code transformations that underly the definitions of types I and II. Using these algorithms, we follow the following process to classify the pairs: first, we 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 clone detectors used in this study do not aim to detect Type IV clones, and therefore we can tag the leftover pairs as Type III pairs.

We verified our algorithms by manually validating 2,800 clone pairs, a random sample of 1,400 pairs taken from each of the Type I and Type II pairs classified by our algorithms. Two judges went through all these pairs independently and reported the algorithms to have perfect precision.

It should be noted that the judges were not verifying if a pair is a clone or not. The judges simply verified if a pair fits the definitions of clone types I and II or not (see definitions in

Listing 3.7: Example of a Type II False Positive Pair

```
1  public GBrowseGFFBuilder(String build, String dbHost, String dbUser, String
       dbPassword, String dbPort, String outputFile) {
2      this.build = build;
3      this.dbHost = dbHost;
4      this.dbUser = dbUser;
5      this.dbPassword = dbPassword;
6      this.dbPort = dbPort;
7      this.outputFile = outputFile;
8  }
9  ---------------------------------
10 public TopicProfile(String id, String name, String siteid, String sitename, String
       unitid, String unitname) {
11     this.id = id;
12     this.name = name;
13     this.siteid = siteid;
14     this.sitename = sitename;
15     this.unitid = unitid;
16     this.unitname = unitname;
17 }
```

Section **??**). For instance, the two code fragments illustrated in Listing 3.7 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 identified as clones, they would be Type II.

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: procedure 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 procedure
```

We discuss the algorithms in the rest of this section.

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 [2]. 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 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 rules, they are categorized as a Type II pair. In order to identify the literals and identifiers for the normalization process, we used an AST parser. The parser failed in parsing some code fragments, and the clone pairs with those code fragments were discarded. The third column of Table 3.9 shows the number of clone pairs that were parseable and were used in this study. The pseudocode for implementing Type II rules is shown in Algorithm 2.

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

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: procedure 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 procedure
```

Type-based Precision Results

In this section, we first describe the process of setting up the type-based experiments. Then, we elaborate on the results we got from these experiments, and discuss how they are different from the undifferentiated results.

Experiment Setup. The setup is similar to undifferentiated precision experiments. With this difference that after separating each tool's clone pairs by type, we made 400 method-level 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 (the tool generating clone pairs) and the type of the clone pairs to remove any bias towards knowing the tools or clone types. Combining the time spent by each judge, in total 80 person hours were spent to complete these experiments.

Results. The results of these experiments are presented in the columns under *Type-based* in Table 3.10. For the sake of comparison, the undifferentiated numbers are also presented. Moreover, to provide an overview of this table, all numbers are also illustrated in Figure 3.6.

The results help us to answer the question – *if a tool's undifferentiated precision is low or high, on which type of clones is its precision low or high?* The earlier precision experiment

Table 3.10: Type-Based Precision

Tool Name	Undifferentiated	Type-based (1200 Pairs Totally)		
		Type I	Type II	Type III
		(400 Pairs)	(400 Pairs)	(400 Pairs)
NiCad	94%	100%	99.25%	93%
SourcererCC	98.5%	100%	100%	98.5%
SimCad	5.5%	100%	100%	6%
iClones	100%	100%	100%	100%
CloneWorks (A)	77.25%	100%	99.25%	74.75%
CloneWorks (C)	95.75%	100%	100%	99.5%
Oreo	82.5%	100%	99.75%	88%
CCAligner	71.25%	100%	100%	67.75%

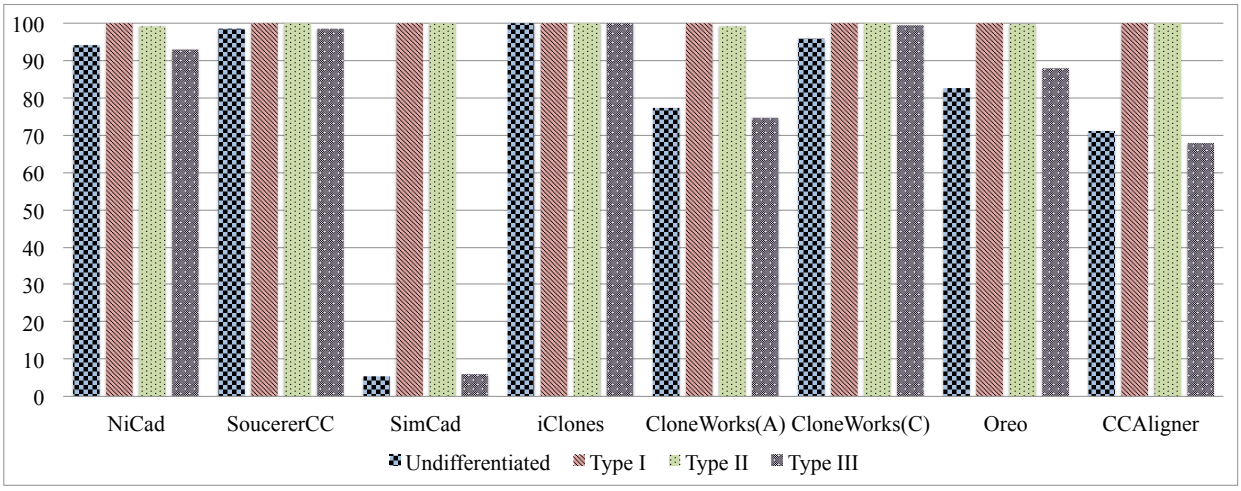


Figure 3.6: Tools' Precision: Undifferentiated and Type-based

results (column *Undifferentiated*) do not represent the performance of tools across all clone types. However, the type-based results show that all tools performed near perfect on Type I and Type II categories. The difference is mostly in the precision of Type III clone category.

For example, consider SimCad. The undifferentiated experiments showed a precision of 5.5% for this tool. This number is pretty low compared to the numbers we got for other tools. However, a closer look at the type-based results shows that SimCad is doing perfect in Type I and Type II clones, and the low precision it gets is only in Type III category. By only looking at undifferentiated results, one cannot gain such an insight, and she/he may argue

that SimCad's results are not as precise as other tools. However, this tool is comparable to others in Type I and Type II, and if one is interested in these types, can safely trust SimCad with respect to precision.

The same fact can be observed for CCAligner, Oreo, and CloneWorks (A). Their undifferentiated precision is not perfect; however, the type-based experiment shows that they have perfect (or near perfect) precision in Type I and Type II categories, and the category that lowers their precision is Type III. These observations raise the need to conduct a type-based precision analysis to gain deeper insight on detection capabilities of clone detectors.

This pattern is not observed for NiCad, SourcererCC, iClones, and CloneWorks (C). As a result, one cannot make a reasoning on the undifferentiated precision numbers and extend it to the precision per type. The type-based precision analysis is needed when the knowledge about the tool's correctness across different types is needed.

Also, with the advent of modern clone detectors such as Oreo and CCAligner that focus on retrieving the harder to get Type III clones (mostly MT3 subcategory), the demand for measuring precision at Type III category is increased. Because the focus of these tools is not retrieving Type I and Type II clones, and they attempt to dive deep into Type III clones, it is important to measure their precision at Type III category.

It is worth mentioning that despite having perfect or near perfect precision and recall for most of the tools in Type I and Type II categories (except for iClones' Type II recall), the number of clone pairs reported by each tool in these types (Table 3.9) is different because of the tools' different settings for clone pairs' sizes; however, in recall and precision experiments, only methods with more than 50 tokens were considered.

3.2.5 Qualitative Analysis

In this section, we provide some qualitative analyses pertaining to the study we conducted. We first discuss how the precision numbers vary between two votes aggregation methods: majority vote and unanimous vote. We provide some qualitative data about these differences, and discuss what can be learned from it. Then, we provide some interesting and harder to judge clone pairs, and argue how precision evaluation task can be affected by the presence of such pairs.

Analysis of Majority Vote and Unanimous Vote

As discussed earlier, the aggregation of judges' votes was done by taking the majority vote. In addition to this method, we also calculated the precision results by taking unanimous voting; that is, judges should be in full agreement about a pair to be true positive for it to be considered as true positive. By comparing the results of the two vote-aggregation methods, we noticed that the numbers do not vary much for Type I and Type II categories, however, they have considerable difference in Type III category. The Type III results using both methods are presented in Table 3.11.

Table 3.11: Precision: Majority Vote vs. Unanimous Vote

Tool Name	Type III	
	Majority Vote	Unanimous Vote
NiCad	93%	78.25%
SourcererCC	98.5%	59.5%
SimCad	6%	3%
iClones	100%	100%
CloneWorks (A)	74.75%	58.5%
CloneWorks (C)	99.5%	83.5%
Oreo	88%	78%
CCAligner	67.75%	55.5%

As it is observed from the table, iClones is the only tool that has gained a perfect score in both methods; this means that the judges did not disagree for any Type III pair of this tool. This pattern, however, does not hold true for the other tools. For instance, using the majority vote, CloneWorks (C) shows a precision of 99.5%. However, with the unanimous vote method, its precision drops by 16%. This gap shows that the judges had dispute in judging some of its pairs. The largest gap belongs to SourcererCC: this tool's precision using the majority vote method is 98.5%, but when using the unanimous vote method, this number drops by 39%. These observations drove us to look deeper into the tools' pairs and understand the disputes among judges. We looked at SourcererCC pairs for which there were disagreements among judges and found out that most of those pairs are *testcases* written to test other methods. These test methods have many occurrences of *assert* statements that cause confusion for judges. One of these pairs is demonstrated in Listing 3.8. One judge argued that: *It is a false positive because most of the arguments given to 'assertEqual' statements are different. So different things are being tested.* The other judge believed this pair to be a true positive and argued that: *both methods are testing based on 'String data'. Despite that two strings are different, yet they are still testing a similar functionality. Also, both methods are making similar method calls on a CsvReader object, indicating they are trying to achieve similar tasks.* The third judge also voted this pair to be a true positive and explained that: *They are both test cases, and key points of them are: assertions, and the input format (both read from csv). If I write a test case for a functionality, I can copy from the first and generate the second.* These reasonings demonstrate the different views that judges have when deciding on a pair, and raise the need to address these kinds of disputes when doing precision studies.

We also looked at a pair from Oreo where judges did not have consensus. This pair is presented in Listing 3.9. Two of the judges marked this as true positive and one judge marked it as false positive. One judge argued that it is a true positive because *both methods iterate through a list and copy the content of the list to another data structure.* The other

Listing 3.8: Clone Pair Reported by SourcererCC

```
1  public void test52() throws Exception {
2      String data = "\\xfa\\u0afa\\xFA\\u0AFA";
3      CsvReader reader = CsvReader.parse(data);
4      reader.setUseTextQualifier(false);
5      reader.setEscapeMode(CsvReader.ESCAPE_MODE_BACKSLASH);
6      Assert.assertTrue(reader.readRecord());
7      Assert.assertEquals("u u ", reader.get(0));
8      Assert.assertEquals("\\xfa\\u0afa\\xFA\\u0AFA", reader.getRawRecord());
9      Assert.assertEquals(0L, reader.getCurrentRecord());
10     Assert.assertEquals(1, reader.getColumnCount());
11     Assert.assertFalse(reader.readRecord());
12     reader.close();
13 }
14 ----------------------------------------
15 public void test3() throws Exception {
16     String data = ",";
17     CsvReader reader = CsvReader.parse(data);
18     Assert.assertTrue(reader.readRecord());
19     Assert.assertEquals("", reader.get(0));
20     Assert.assertEquals("", reader.get(1));
21     Assert.assertEquals(',', reader.getDelimiter());
22     Assert.assertEquals(0L, reader.getCurrentRecord());
23     Assert.assertEquals(2, reader.getColumnCount());
24     Assert.assertEquals(",", reader.getRawRecord());
25     Assert.assertFalse(reader.readRecord());
26     Assert.assertEquals("", reader.getRawRecord());
27     reader.close();
28 }
```

Listing 3.9: Clone Pair Reported by Oreo

```
1  private void createSnippetIDs(List<TreeNode> snippets) {
2      int currentID = 0;
3      Iterator<TreeNode> iterator = snippets.iterator();
4      while (iterator.hasNext()) {
5          TreeNode snippet = iterator.next();
6          String snippetID = "snippet_" + currentID;
7          snippetIDs.put(snippet, snippetID);
8          ++currentID;
9      }
10 }
11 ---------------------------------
12 protected void setExportElmClassPath(ArrayList<String> cp) {
13     String cps[] = new String[cp.size()];
14     Iterator<String> i = cp.iterator();
15     int j = 0;
16     while (i.hasNext()) {
17         cps[j] = i.next();
18         j++;
19     }
20     exportElmClassPath = cps;
21 }
```

judge said: *both methods iterate on the input parameter. The top one uses currentId to create a [key,value] pair for each item in the iterator. It then puts the pair into a HashMap and increases currentId. The second method also does the same, increases j and uses j as a key (index) to store items in an array. Hence, I vote for true positive.* However, the third judge argued that: *The only common part of the two methods is that both iterate over a list and store the contents of this list into another. However, the upper one builds a special 'snippetID' and stores that in a HashMap, whereas the bottom one stores the content of input list, as it is, in another array. I vote for false positive here.*

These examples, and the gaps between the majority vote and unanimous vote methods, show that precision experiments are greatly dependent on human judges' perception of clones, and that judges can have different opinions when judging the same pair. Hence, there is a need for these precision studies to be accompanied by a report on *inter-rater agreement* [2, 117]. Such a measurement would give us information as to the percentage of pairs for which judges had

consensus in their judgments. This measurement is an indicative of the confidence we can have on judgments. A conservative measurement of inter-rater agreement can be calculated by taking the proportion of times judges have the same vote. In our experiments, the proportion of times the judges had the same votes (all true positive or all false positive) for undifferentiated experiments is ≈86% (2,745 pairs out of the total of 3,200 validated pairs), and for type-based experiments is ≈92% (8,794 pairs out of the total of 9,600 validated pairs).

Analysis of Interesting True Positives

While conducting the precision experiments, judges noticed that there exist some pairs that are true positives, but, at the first sight, they looked like a false positive to them. A closer look was needed to understand that these pairs are true clone pairs. We present and analyze two of those pairs here. The first instance is presented in Listing 3.10. Both methods in this example first make sure that their input parameter is not *null*. The upper one, then, looks for the index of dot in the input string; if dot is not found (the returned index is -1), then the method returns null. Otherwise, it returns a substring of the input string from its beginning until the location of the found dot. A close look at the bottom method shows that this method is also performing the same functionality, except that instead of dot, it operates on a character that is stored in *URI_SEPARATION_CHAR*.

Another example of this kind is illustrated in Listing 3.11. The upper method in this example receives an array of int as input, and then converts it to an array of double. The bottom one also implements the similar functionality: instead of an int array, it receives an array of BigDecimal values, and then stores its values into an array of double. The upper method does not have any explicit type casting. The bottom method, however, includes class cast operation to convert to double. This casting is done by invoking the *doubleValue()* method.

Listing 3.10: True Positive Clone Pair: Example1

```
public static String getGroupId(String layoutId) {
    if (layoutId == null) {
        return null;
    }
    int pos = layoutId.indexOf(".");
    if (pos == -1) {
        return null;
    } else {
        return layoutId.substring(0, pos);
    }
}
-----------------------------------
public static String parseFullTargetObjectURI(String fullTargetObjectURI) {
    if (fullTargetObjectURI == null) return null;
    int delimIndex = fullTargetObjectURI.indexOf(URI_SEPARATION_CHAR);
    if ((fullTargetObjectURI != null) && (delimIndex != -1)) return
        fullTargetObjectURI.substring(0, delimIndex); else return fullTargetObjectURI;
}
```

The two discussed examples are instances where clone pairs do not share much of syntactic similarity, but semantically are very similar. Presence of such clone pairs indicates that precision studies are more complicated than comparing two program texts. They need deep analysis of program behavior and understanding the goal of each code fragment in a pair. Hence, precision analyses need careful consideration of judges, and researchers should be aware of this.

3.2.6 Related Work

Several studies have been done with the aim of analyzing and evaluating clone detection tools. In this section, I explain the ones that are most related to ours, and discuss how our work is different from them.

Svajlenko et al. evaluate ten clone detection tools proposed until 2015 with respect to their recall [121]. The dataset used in this study is the portion of IJa dataset 2.0 used by

Listing 3.11: True Positive Clone Pair: Example2

```
public static double[] int2double(int [] v) {
    double[] ia = new double[v.length];
    for (int i = 0; i < v.length; i++) {
        ia[i] = v[i];
    }
    return ia;
}
--------------------------------------------------
public static double[] bigDecimalArrayToDoubleArray(BigDecimal[] bigDecimalArray) {
    double[] doubleArray = new double[bigDecimalArray.length];
    for (int i = 0; i < bigDecimalArray.length; i++) {
        doubleArray[i] = bigDecimalArray[i].doubleValue();
    }
    return doubleArray;
}
```

BigCloneBench, the same dataset as the one used in our study. BigCloneBench has more than 8 million clone pairs tagged. However, not all possible clone pairs are tagged in this dataset, and there exists many untagged clone pairs. In another work, Svajlenko et al. state that BigCloneBench can be used to measure an upper bound and a lower bound for precision [120]; however, as stated by them, this range can be quite wide. They also propose a formula that estimates the precision, only based on the known clone pairs in BigCloneBench and ignoring the unknown clone pairs. However, ignoring the many clone pairs that are not tagged in BigCloneBench cannot provide a realistic estimation for tools' precision.

The other study focused on evaluating clone detection tools is the one conducted by Bellon et al. [19]. They have evaluated six clone detectors, proposed until 2002, on eight C and Java programs (altogether 850 KLOC). One author of the paper, Stephen Bellon, has created a dataset of known clone pairs by going through 2 percent of the 325,935 pairs reported by the six tools, and manually validating this set of pairs. Then, precision and recall of tools have been estimated against the validated clone pairs in the dataset. There are some limitations with this study; first, although manually validated, the tagged clone pairs are reported by the same set of six tools. This may impose some bias in numbers. Second, precision is

estimated based on the validated pairs, and as stated by the authors, their numbers are relative and should be used with caution.

A more recent study is the one carried out by Ragkhitwetsagul et al. [95]. They have evaluated 30 code similarity detection techniques and tools using five experimental scenarios. This study is aimed at similarity detection tools, and hence includes a wider spectrum of tools than clone detectors. 5 of the target tools are clone detection tools, and others fall in other categories: Obfuscutors, compilers and decompilers, plagiarism detectors, compression tools, and one category named as *other*.

Another work on studying clone detection tools has been done by Roy et al. [105]. They provide a qualitative comparison on the state of the art tools presented until article's publish date (2009). They define a taxonomy for clone detection tools, based on the level of analysis that clone detectors apply to the source code. The state of the art tools are categorized based on this taxonomy, and they are also compared based on a number of facets such as language facet, clone facet, and code representation facet. Finally, tools are analyzed based on four different scenarios.

In addition to the studies dedicated to evaluating clone detection tools, these tools have been compared in studies where a new clone detection tool is proposed. Oreo [108] and SourecerCC [111] are two examples that have conducted manual analyses to measure the precision of their tools, and other state of the art tools. Both of these have estimated precision by taking a statistically significant sample of clone pairs. However, these studies do not provide a type-based precision study, and are limited to providing undifferentiated numbers.

Our study is different from the mentioned studies in that it focuses on precision. Also, while other work that has reported precision, have followed the undifferentiated method, we did both undifferentiated and type-based analysis. We manually validated statistically

significant samples of the tools' clone pairs for both methods. We presented results of both experiments, compared the results, discovered the issues, and provided insights to help the future work. Also, apart from using the state of the art near miss Type III clone detectors, we include two recently published clone detectors, Oreo and CCAligner, which detect harder to get Type III clones.

3.2.7 Threats to Validity

The separation of clone pairs based on types was done using certain algorithms. Although it was done by following the definitions of clone types, there maybe errors in the results. We managed this issue by evaluating sets of sampled clone pairs by two judges. The results demonstrated a 100% precision for the algorithms.

The target tools of this study have various configurations, and each can result in different set of detected clone pairs. We studied the tools' papers, related literature, and also, contacted the tools' authors to configure the tools so that we can match the recall numbers reported by corresponding papers as much as possible. The results presented in this study are valid for the tools' configuration provided here, and they may not hold true if other configurations are used.

Precision evaluation is a manual task that can be subject to human bias. We reduced this bias by anonymizing the tools' clone pairs when presenting to judges. Also, in the type-based experiments, we merged and shuffled all types' clone pairs, and presented a single set of clone pairs for each tool to users. This alleviates the bias pertaining to being aware of clone types.

3.2.8 Conclusions and Future Work

In this study, we presented systematic experiments to measure the precision of eight clone detection tools using two methods: undifferentiated and type-based. Both ways of conducting precision experiments involve a large amount of manual work. The undifferentiated experiment is faster than the type-based one because there are less number of pairs to validate. The undifferentiated experiment gives an idea of the overall precision of the tool, without considering clone types. This precision is useful to understand how many true positives one could expect to see on an average for every 100 pairs in the output of a clone detector. The type based precision experiment, however, reveals information about the effectiveness of a given tool for different clone types. This experiment, along with the undifferentiated experiment, gives a better understanding of the effectiveness of target clone detectors and therefore, both experiments should be conducted if one wants to compare different clone detectors. This is particularly important now that the detection of simpler types of clones (types I and II) is essentially a solved problem, as attested by our results. Cutting edge research is moving on to harder-to-detect Type III clones, and even Type IV. Comparative results can be made more fair and enlightening by including clone type information in precision measurements, and/or ignoring types I and II entirely.

Evaluating clone pairs is a subjective task, and the degree of agreements on judgments is an important factor to consider. Hence, measuring *inter-rate agreement* is a factor that if accompanied by precision numbers, can shed lights on the reliability of precision results. Finally, clone pair validation is a complex task that needs knowledge and expertise. A person in charge of conducting such experiment needs to be careful of cases where syntactic similarity is not the basis for clone evaluation; rather, it is the semantic, or functional, similarity that needs to be analyzed.

We aim to continue this work by diving deeper into analyzing the precision in clone categories;

that is, to separate the Type III clones into their subcategories (VST3, ST3, MT3, and WT3/T4), and then analyze the precision in each of these subcategories. Such an analysis would give us a broader view of the effectiveness of the tools. Moreover, in future, we will analyze the precision of tools that work on languages other than Java, and conduct comparative studies on those tools.

Chapter 4

Similarity Detection of Neural Network Models with RICA

4.1 Introduction

This chapter formulates the problem of DNN functional similarity detection and presents RICA (Random Inputs and Correlation Analysis), as a method for detecting functional similarities among DNN models. The key point about RICA is that it works without having any knowledge about the DNN models being analyzed, and does not need to have access to models' training/testing data or training scripts; it works by only accessing the models' files.

RICA works based on the insight that given two models, their similarity can be measured by comparing their outputs on the same set of inputs using an appropriate similarity metric. This intuitive statement about model similarity, however, needs to be operationalized in two fronts: (1) what inputs should be used to assess similarity, and (2) how can we quantify "sufficiently similar"? With respect to (1), test sets are the obvious candidates, but they may not always be available. In fact, in some cases we may not even know what the models

are supposed to do, much less what data was used to train and test them. With respect to (2), methods and metrics that have been used for comparing traditional code, for example in assessing functional equivalence [52], fall short of capturing similarity of DNNs. Recent work in this area focuses on statistical methods and metrics to compare the representations learned by DNNs at various layers, using meaningful inputs. One promising family of methods are based on the Canonical Correlation Analysis (CCA) [46], which has been recently applied to neural networks [94, 86]. Another promising, but more constrained, metric is the Spearman rank correlation, which has been applied in a small study using meaningful canonical inputs [119]. Finally, for classifiers in particular, we can use a simple overlap metric that quantifies how many times two different models agree on the classification of the same inputs.

RICA uses random inputs in lieu of canonical inputs to perform models' input/output analysis and is capable of detecting functionally similar DNN models using three metrics: *CCA, Spearman*, and *Overlap*; the latter only being applicable to classification models. In the rest of this chapter, I will elaborate on the problem of DNN functional similarity detection and will also explain how RICA uses random inputs with the three similarity metrics discussed above to detect DNN clones.

4.2 Problem Definition

Here, I first define the concept of models' functional similarity, and how it can be quantified. This is followed by a discussion of challenges of DNN models clone detection.

4.2.1 Functional Similarity

Functional similarity measures the extent to which the functions performed by two models are similar. Unlike traditional code, comparing the training and testing scripts of models is useless for the purposes of model similarity since the functionality of DNN models is mostly based on the data that is used for training them; it is common to have identical training scripts that produce entirely different DNN models. Therefore, detecting model similarity requires actual input/output analysis.

Before defining model similarity, we define model *equivalence*. In traditional code, functional equivalence has previously been defined as the equivalence of output given the same input accounting for permutations of inputs and outputs [52]. We generalize that definition for DNNs. Let m_1 and m_2 be two DNN models; let I be a set of inputs for m_1. We say that m_2 is functionally *equivalent* to m_1 if $m_2(\phi(i)) = m_1(i), \forall i \in I$, where $\phi(i)$ is a geometric transformation [21] of the input, which could be the identity map, but also scaling, reflection, projection and other such transformations.

Functional equivalence, however, is of limited interest. Models can be very similar without producing the exact same outputs for the same (or transformed) inputs. Consider, for example, two models, each with one single numerical output, which, for the same inputs produce outputs such that $o_2 = 2o_1, \forall o \in O$ – i.e. the second model's output values are always double the first model's output values; clearly, these models are doing something very similar even if the values are scaled by a factor of 2. As a second example, consider two models which, for the same 10,000 inputs, produce the exact same outputs in 9,000 cases, and different outputs in 1,000 cases; again, these two models are similar, even if their outputs differ 10% of the time.

Given this, similarity of DNN models is best captured by measuring *correlation* between the outputs [119, 94, 86]. If for the same (or transformed) inputs, the outputs of two models

are correlated, then the two models are functionally similar. Formally, m_1 and m_2 are functionally similar if

$$Corr(m_1(i), m_2(\phi(i))) > \theta, \forall i \in I \tag{4.1}$$

where $Corr$ is a correlation metric, ϕ is a geometric transformation, and θ is a similarity threshold.

In the case of models with more than one output, and even with different number or ordering of outputs, the correlation metric must reflect an aggregation of the several outputs using some aggregation method. In this general definition, $Corr$ stands for the aggregated correlation of the outputs.

4.2.2 Challenges of DNN Clone Detection

Consider the two functions in Listing 4.1. The first function, spanning from lines 2 to 10, creates and trains a model on the MNIST dataset [77], and the second function, from line 12 to 21, creates a model and trains it on the Fashion-MNIST (FMNIST) dataset [143]. In terms of code, they only have one token difference: on the second line of each method where the training data is loaded, one uses the token *mnist* and the other uses the token *fashion_mnist* to load the training data. Other than this subtle difference, the two functions are identical, and set up the exact same network architecture. A traditional code clone detector would signal these functions as clones (specifically, Type-2 clones [105]). However, the models trained by these functions address entirely different tasks since each is trained on a different dataset: one classifies hand-written digits and the other one classifies pieces of clothing. By only analyzing the code that sets up the models, or the network architecture itself, one cannot reason about the functional similarities and differences between the models themselves.

```
def train_my_model1():#train on mnist
    import keras
    (X_train,Y_train),(X_test,Y_test)= mnist.load_data()
    X_train=(X_train.reshape((60000,28*28))).astype('float32')/255
    Y_train=keras.utils.to_categorical(Y_train)
    dnn=keras.models.Sequential()
    dnn.add(keras.layers.Dense(512,activation='relu',input_shape=(28*28,)))
    dnn.add(keras.layers.Dense(10,activation='softmax'))
    dnn.compile(optimizer='rmsprop',loss='categorical_crossentropy',metrics=['accuracy'])
    dnn.fit(X_train,Y_train,epochs=10,batch_size=128)

def train_my_model2():#train on fashion_mnist
    import keras
    (X_train,Y_train),(X_test,Y_test)= fashion_mnist.load_data()
    X_train=(X_train.reshape((60000,28*28))).astype('float32')/255
    Y_train=keras.utils.to_categorical(Y_train)
    dnn=keras.models.Sequential()
    dnn.add(keras.layers.Dense(512,activation='relu',input_shape=(28*28,)))
    dnn.add(keras.layers.Dense(10,activation='softmax'))
    dnn.compile(optimizer='rmsprop',loss='categorical_crossentropy',metrics=['accuracy'])
    dnn.fit(X_train,Y_train,epochs=10,batch_size=128)
```

Listing 4.1: Two almost identical training scripts

As this example shows, the behavior of a DNN model is heavily reliant on the dataset it was trained on. Therefore, models' functional similarity cannot be inferred from analyzing the training code. Functional similarity assessment demands an analysis of training data itself, or analyzing models' behavior using canonical datasets (typically training or testing data). Such data, as mentioned earlier as well, might not be available as a part of automatic clone detection (for instance, due to security or privacy reasons [89] or simply due to the size of the data). All these issues make models clone detection a challenging process that cannot be carried out using traditional clone detection techniques; therefore, needing special methods built specifically for this purpose.

4.3 Goals and Scope

The definition of functional similarity given in Equation 4.1 requires the existence of inputs for the purpose of input/output analysis. In general, meaningful canonical inputs, such as training or testing data, may not be available. Therefore, we need to have suitable inputs

at hand in order to perform the analysis. RICA utilizes random inputs for this purpose. The other aspect of this comparison is the selection of an appropriate metric to quantify the similarity. That is to define how to compare the outputs of two models over random inputs and how to decide if two models are functionally similar. To this aim, we study a set of previous metrics that were used in the literature for the purpose of similarity assessment when canonical inputs are available. We investigate the applicability of these metrics when random inputs are used and the implications of using each of these metrics.

As such, with proposing RICA, we address two primary goals: (1) we investigate whether random inputs can be used, instead of canonical ones, to detect model similarity; and (2) we study the trade-offs of different similarity metrics when using random inputs. Our findings shed light on the process of model similarity detection and on the applicability of individual metrics in different scenarios.

Equation 4.1 is generic, in the sense that it can be used to compare any two DNN models independent of their nature and input/output shapes and dimensions. In this work, we limit our attention to models that have compatible input shapes and the same output shapes. This helps us limit the scope of our work and define the scope of models' comparison. Therefore, RICA, as it currently stands, can be used in such scenarios only.

Compatible input shapes here means that the input shapes of two models m_1 and m_2 can be derived from each other by simple reshape transformations. The reason for considering compatible input shapes in this way is that developers oftentimes reshape their inputs and create models based on the resulting input shape. For example, in case of the MNIST dataset, the raw data consists of images sized 28×28. A developer may choose to create a model that receives its input in this shape, or may flatten the input into a vector of 784 numbers, or may decide to add the channel information to the input by reshaping it to $28 \times 28 \times 1$ and create a convolutional neural network (CNN). All these models perform MNIST image classification, so we need to be able to compare them. The same can happen when a model

has more than one input layer, and, for example, receives two vectors as input; it is possible to train a similar model on the same dataset by passing one input that includes the contents of the two vectors concatenated in one. For this reason, input shape compatibility needs to go beyond exact shapes: given two models, we check whether their input shapes can be converted to one another.

Furthermore, we limit our primary attention to classifiers. Since we limit our scope to models that have the same output shapes, in case of classifiers that are the main focus of our work, classifiers with different number of output classes, such as one with 10 classes and another with 11, are out of scope. In our evaluations, we narrow the scope of this study to single-label multi-class DNN classifiers with compatible input shapes and the same number, and ordering, of output classes. However, later in Chapter 5, we show that the approach followed by RICA can also be used to assess the similarity of regression models.

These constraints were followed simply to tame the complexity of the analysis, as trying to compare any two models would not only increase the dataset considerably but would also substantially expand the results and analysis.

4.4 Design of RICA

4.4.1 Similarity Inspection Pipeline

The pipeline designed to perform similarity detection with RICA is shown in Figure 4.1. The pipeline consists of two high-level modules: (i) a compatibility verification module and, (ii) a functional similarity module. The compatibility verification module investigates whether the two models being analyzed satisfy the assumptions of our approach regarding input and output shapes compatibility. Functional Similarity module, then, performs the

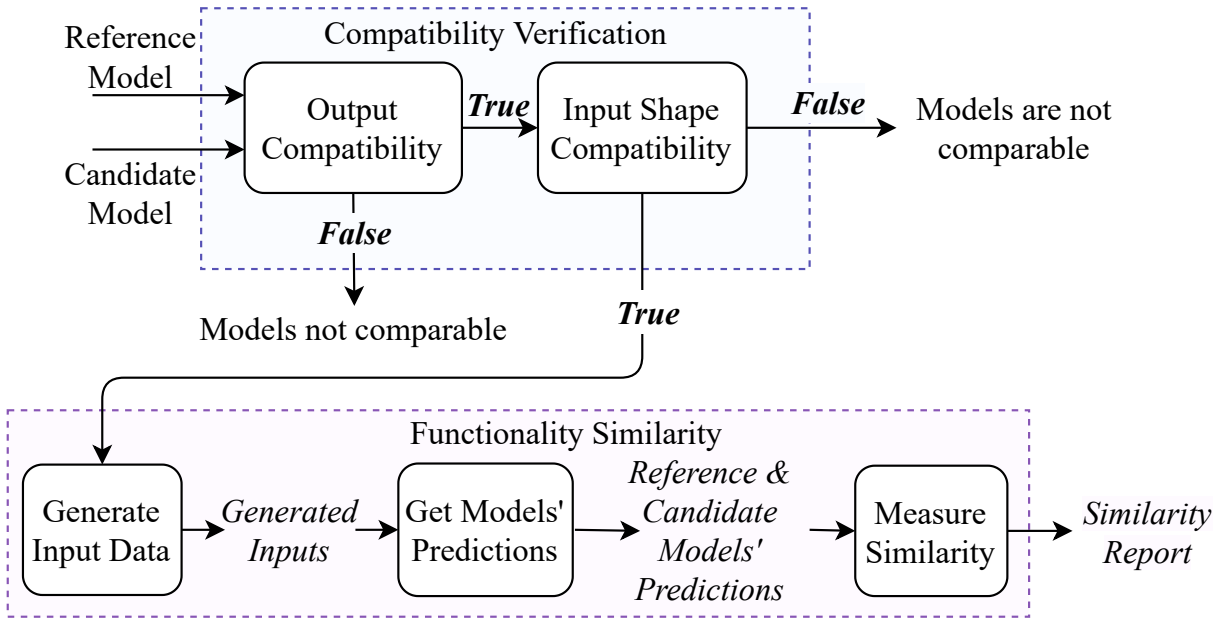


Figure 4.1: Overview of the similarity detection pipeline by RICA

main comparison producing a similarity score that can be interpreted using a set of defined thresholds depending on the similarity metric being used (explained later). The details of each of the modules are described below.

Compatibility Verification

In order for two models to be considered comparable with RICA, they need to be compatible in terms of the input and output shapes. The compatibility of two models is verified as explained below:

Output Compatibility. Two models have output compatibility if their output shapes (including the activation function of the last layer) are the same. In case of classifiers, which are the main focus of this work, models have output compatibility iff $n_r == n_c$ and they have the same activation function in their output layers. In this definition, n_r is the number of output labels of the reference model and n_c is the number of output labels of the candidate model. The algorithm followed to investigate output compatibility is shown in Algorithm 3.

First (lines 2 and 3), the activation function of the output layer of both models is retrieved, followed by extracting the number of output labels for both models (lines 4 and 5). The algorithm returns true if both the activation functions of the output layers and the number of output labels for the two models are equal; otherwise, it returns false.

Algorithm 3 Output compatibility

```
1: procedure isOutputCompatible(model_Reference, model_Candidate)
2:     activ_Reference ← getOutputActivation(model_Reference)
3:     activ_Candidate ← getOutputActivation(model_Candidate)
4:     numLbl_Reference ← getNumberOfLabels(model_Reference)
5:     numLbl_Candidate ← getNumberOfLabels(model_Candidate)
6:     if numLbl_Reference == numLbl_Candidate and activ_Reference == activ_Candidate then
7:         return True
8:     else
9:         return False
10:    end if
11: end procedure
```

Input Compatibility. Input shape compatibility verifies whether the input shapes of two models can be derived from one another. To perform this investigation, we flatten both input shapes into a single-dimension vector. In case of models with more than one input, each input is first flattened, and then they are all concatenated. Two models have compatible input iff $length(flat(i_r)) == length(flat(i_c))$ where i_r is the reference model's input shape and i_c is the candidate model's input shape. Algorithm 4 shows the detailed steps of input compatibility verification.

Algorithm 4 Input compatibility

```
1: procedure isInputCompatible(model_Reference, model_Candidate)
2:     flatInputShape_Reference ← getFlatInputShape(model_Reference)
3:     flatInputShape_Candidate ← getFlatInputShape(model_Candidate)
4:     if flatInputShape_Reference == flatInputShape_Candidate then
5:         return True
6:     else
7:         return False
8:     end if
9: end procedure
```

Functional Similarity

As Figure 4.1 shows, functional similarity is investigated by first generating a set of input data. The mechanics of data generation step can be different depending on the similarity metric being used, which will be detailed later. Regardless of the similarity metric, however, the input data contains float values between -1 and +1. This step is followed by getting both models' predictions on the generated inputs, and then measuring the similarity on the generated inputs using each metric. The details of the similarity metrics that can be used here, along with the details of the usage of each of them and their thresholds for deciding about models' similarity/dissimilarity is explained through the rest of this section.

4.4.2 Functional Similarity Metrics

To assess the functional similarity between two models, we analyze their predictions on the same random inputs. Since the main focus of this work is on classifiers, we will focus our explanation on the case of classifiers. In order to quantify similarity, we consider three metrics: Canonical Correlation Analysis (CCA), Spearman rank correlation (ρ), and a simple overlap metric. The choice of these metrics is based on previous similarity studies: CCA has been used for measuring the similarity between the representations learned by neural networks in their intermediate layers [94, 86], and Spearman correlation is a suitable similarity metric [3] that has been used in assessing models' functional similarity when canonical inputs are available [119]. The overlap metric is inspired by the study of code functional equivalence by Jiang et al. [52], adapted to measure similarity; here we assume that the order of the outputs is the same, and simply measure the degree of overlap on models' outputs given the same inputs. Details of each of these metrics are explained below.

CCA: CCA is a statistical method for inferring the relationship between two sets of variables, by finding linear relationships between them such that the correlation between the

linear relationships is maximized [119, 86]. CCA has previously been used in measuring the similarity between the learned representations of different layers of different neural networks by feeding canonical datasets to the networks and measuring CCA on the activations of these layers [94, 86]. CCA is very generic, and is capable of finding the relationship between two sets of variables with different cardinalities, making it a viable choice in assessing the functional similarity where the models' output dimensions are different [119].[1] The correlation coefficient in CCA ranges from 0 to 1 [116]. In order to use CCA to measure the similarities among classifiers, RICA calculates the CCA correlation coefficients over the output prediction probability vectors of the two models over the m inputs fed to them. This produces n correlation values $\{Corr^1, Corr^2, .., Corr^n\}$ where n is the number of output labels and $Corr$ is the computed correlation value using CCA. The results are then aggregated by calculating the *mean* value over the n correlation values, similar to the method explained in [86]. This mean value serves as the final similarity value used to assess the degree of similarity between two models.

Spearman: Spearman correlation [119] is a non-parametric correlation metric (unlike Pearson correlation, for example) [3]. It has been used in previous studies of neural network functional similarity using canonical inputs [119][2]. Spearman correlation coefficient ranges between -1 and 1, where positive values denote a direct correlation, negative values denote an inverse correlation, and values close to zero (typically between -0.1 and 0.1) denote the absence of correlation. To use Spearman correlation as a similarity metric, RICA considers the probability values reported for each output label by each of the two models as one variable and calculates the correlation values between the corresponding outputs of the two models across all m samples. In other words, if model m_1 has n output classes $\{c_1^1, c_1^2, .., c_1^n\}$

[1] In the case of our work, and as explained before, the output dimensions of comparable models are never different.

[2] We also considered using Kendall's Tau correlation, which is also non-parametric [3]. However, we did not observe much difference between the correlation values calculated with these two metrics, and therefore, chose to use the Spearman correlation, as it is computationally faster ($O(mlog(m))$ compared to Kendall's Tau ($O(m^2)$) [3].

and model m_2 also has n output classes $\{c_2^1, c_2^2, .., c_2^n\}$, RICA calculates n correlation values $\{\rho^1, \rho^2, .., \rho^n\}$ such that $\rho^i = Correlation(c_1^i, c_2^i)$. To aggregate the results, RICA computes the *mean* of these n correlation values and uses this mean value as the final similarity value. Using this metric, therefore, requires the two models to have the same output shapes and dimensions, and to interpret them in the same order, which, as explained before, is a simplifying assumption of our study.,

Overlap: Finally, the last and the simplest similarity metric that is used by RICA is a simple *overlap*, i.e. the number of times the two models agree on the classification of the random inputs. Given its categorical nature, this metric is only applicable to classifiers.

In summary: CCA is the most generic metric, as it can be used to compare any two models; our use of Spearman correlation assumes models with the same number and order of outputs; and the simple overlap metric assumes that models are classifiers.

4.4.3 Similarity Thresholds

The similarity metrics need to have a *threshold* at which we classify models as similar or dissimilar. Here, we explain the thresholds used for each of the metrics by RICA.

CCA and Spearman

For CCA and Spearman correlation, we can tune the threshold empirically using the general and accepted guidelines for correlation ranges: a value between 0 and 0.1 is generally considered no correlation, a value between 0.1 and 0.2 is considered weak positive correlation, between 0.2 and 0.5 is moderate and positive, and more than 0.5 is a high positive correlation. For the purpose of similarity, we consider moderate and above correlation values (values $>= 0.2$) as similar, weak correlation values ($0.1 <$ values < 0.2) as uncertain, and

values <= 0.1 as dissimilar. For Spearman metric that reports inverse correlations with negative values, we consider those as dissimilar as well, since an inverse relationship also shows differences between the predictions. The similarity and dissimilarity regions are summarized below. Here, S shows the final similarity value calculated using either CCA or Spearman.

$$Regions: \begin{cases} \textit{Dissimilarity Region}, & \text{if } S \leq 0.1 \\ \textit{Uncertainty Region}, & \text{if } 0.1 < S < 0.2 \\ \textit{Similarity Region}, & \text{if } S \geq 0.2 \end{cases}$$

Overlap

For the overlap metric, the calculated similarity values show the fraction of times the two models predict the same label over the same input. This metric is similar to accuracy where the number of times the predicted labels match the ground truth labels is counted; in our case, the ground truth is the reference model's predictions and we count the number of times the candidate model's predicted labels match those of the reference model. One important note here is that accuracy has been shown to be negatively affected if the underlying input dataset has an unbalanced distribution of labels [84]. This problem is extended to the Overlap metric as well: when the reference model's predicted labels over the inputs have an unbalanced distribution, the calculated Overlap values become unreliable and it will be impossible to specify a similarity/dissimilarity threshold that works for all kinds of classifiers, with any number of labels.

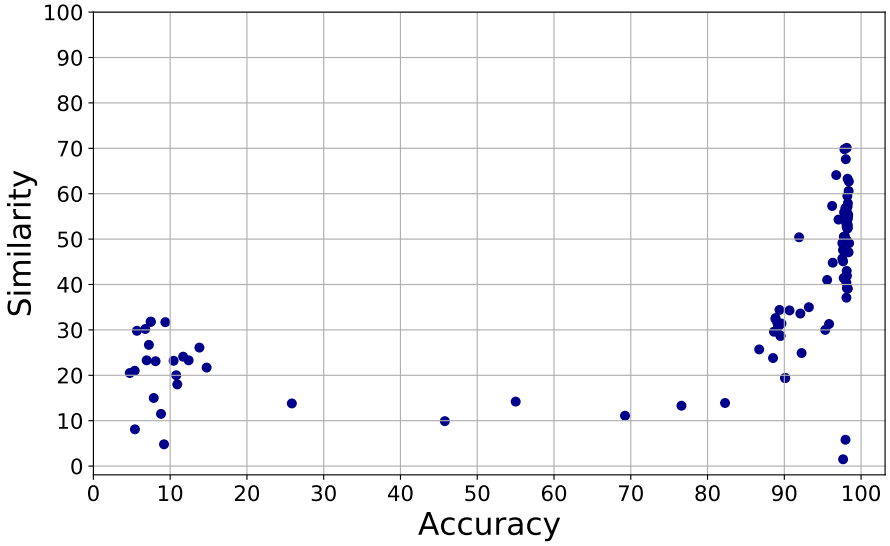
The aforementioned issue is illustrated in the two plots of Figure 4.2. These plots show the overlap similarity predictions (y-axis) for two similar MNIST digit recognition models against many others using the same set of unbalanced random inputs. The x-axis here

shows the candidate models' accuracy on the reference model's testing dataset, therefore, being the ground truth of similarity (the intuition here is that if the candidate model has a high accuracy on the testing dataset of the reference model, then the two models have a high chance of being functionally similar). As we see in the plots, in spite of the two reference models being similar with each other, the similarity predictions calculated for them with the other models are very different, and, even worse, they are scattered. This makes it difficult to decide where to draw the threshold line to decide about the similarity. The situation is even worse with Model2 where the similarity measurements for both similar and dissimilar models reach $\approx 40\%$. Any chosen line can result in several false positives.

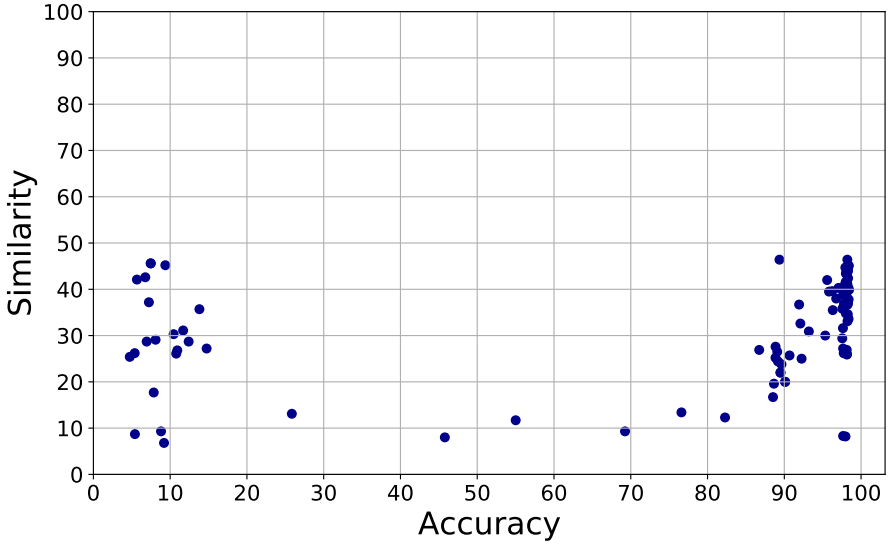
This issue is mainly caused by the unbalanced distribution of labels by the two models. It is possible that the generated unconstrained random inputs result in a major coverage of one (or more) of the labels by the reference model, while the rest of the labels are covered only a few times (due to models' biases for example). Similar to the case of accuracy, this makes the overlap values unstable and unreliable.

In practice, we observed that with unconstrained generation of random inputs, these inputs end up being biased towards a set of labels, therefore, endangering the specification of similarity thresholds for the Overlap metric. Figure 4.3, for example, shows the label coverage that a model trained on the MNIST dataset achieves on an unconstrained set of random inputs. In this case, label 5 dominates the label coverage, and labels 0 and 8 appear very few times. This motivates the next option for the input generation process: to impose some constraints to the input generation process so that the *reference model's predicted labels on the generated random inputs have a balanced distribution.*

The two plots shown in Figure 4.4 show the similarities for the same two models discussed in Figure 4.2 on balanced inputs. Here, the similarity values are clustered, and it is possible to establish a threshold that fits both cases without false positives (the similar models have predicted similarity values above $\approx 20\%$).



(a) Model1: Random Inputs



(b) Model2: Random Inputs

Figure 4.2: Overlap metric: random unconstrained inputs

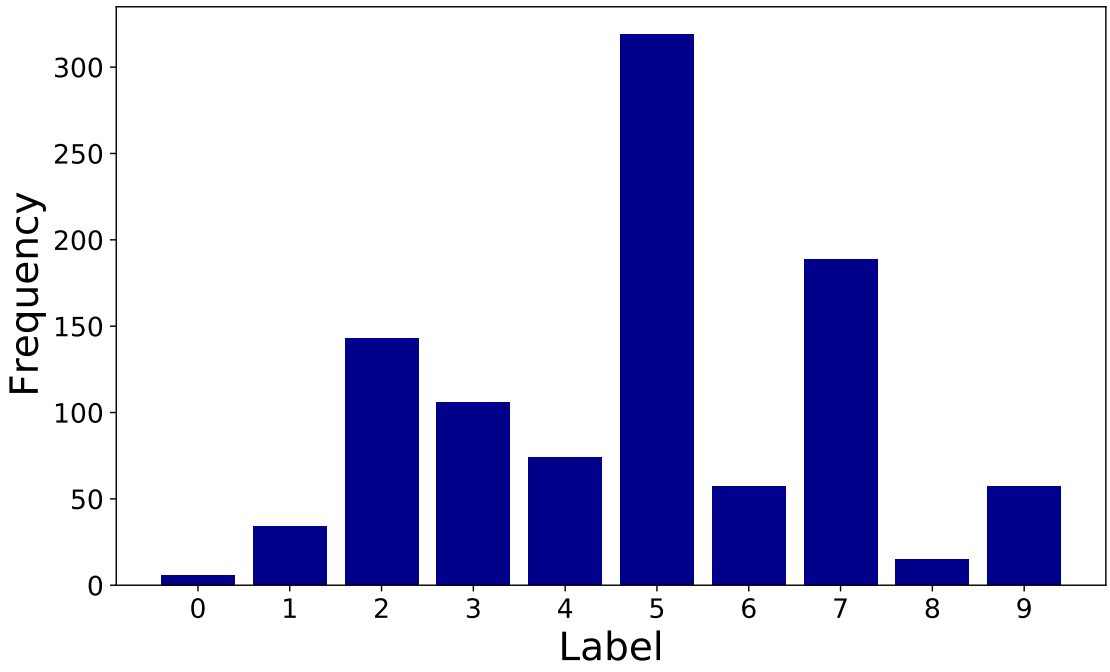
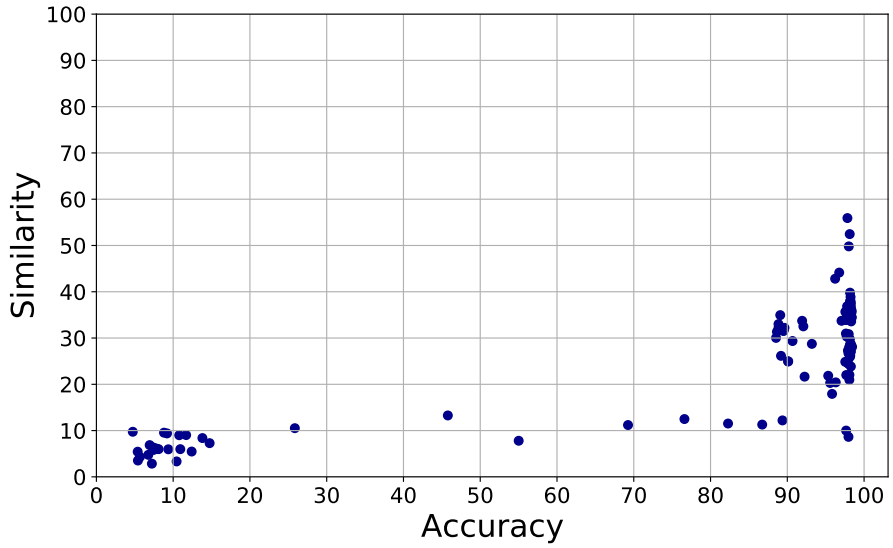
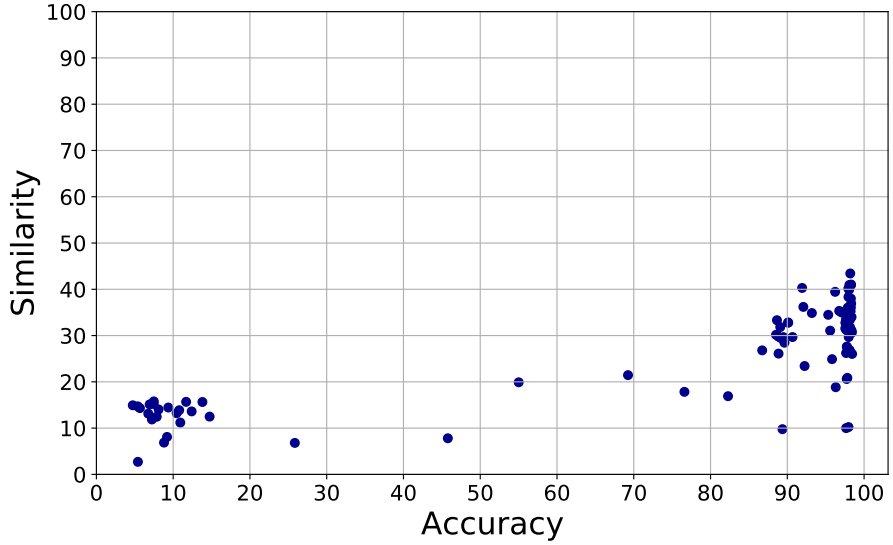


Figure 4.3: Frequency of labels with Unconstrained Random Inputs

In particular, if the input dataset d has m rows and is balanced with respect to the reference model m_r which has n_r output classes (that is m_r predicting each output class for $\frac{m}{n_r}$ times on d), then any arbitrary model, just by pure chance, can agree with m_r for $1/n_r$ times. Therefore, when the level of agreements between m_r and another model using the dataset d is at the level of chance ($\lessapprox 1/n_r$), the two models cannot be classified as similar. If the agreements level is much higher than $1/n_r$, for example twice the level of chance ($2/n_r$), then something other than chance is at play and is a strong indicator that the models are doing something similar. The upper threshold of $2/n_r$, however, can be too strict given model's imperfect accuracies. In the case of binary classifiers, for example, this threshold translates to 100% similarity which does not happen in practice. Even the most similar models do not agree with each other 100% of times, and their accuracies affect their agreements level. In general, we do not know the models' accuracies, but we know that a 100% accuracy is rare. Therefore, the similarity threshold $2/n_r$ should be multiplied by an empirical factor α, and become $2\alpha/n_r$. For most good models, an accuracy above 90% is expected and therefore,



(a) Model1: Balanced Inputs



(b) Model2: Balanced Inputs

Figure 4.4: Overlap metric: balanced inputs

depending on whether both models' accuracies are taken into account or only one of them, α can have a value between 0.81 and 0.9. In our experiments, we observed that $\alpha = 0.9$ works well while still eliminating the false positives.

Based on this, when using the Overlap metric, we generate *balanced random inputs*, where with n number of output classes, a similarity $<= 1/n$ is considered dissimilar, a similarity $>= 2\alpha/n$ is considered similar, and between these two is uncertain where we cannot comment on models' similarity with confidence. The similarity and dissimilarity ranges for this case are summarized below. Here, S demonstrates the final similarity value calculated with the overlap metric.

$$Regions: \begin{cases} \textit{Dissimilarity Region}, & \text{if } S \leq 1/n \\ \textit{Uncertainty Region}, & \text{if } 1/n < S < 2\alpha/n \\ \textit{Similarity Region}, & \text{if } S \geq 2\alpha/n \end{cases}$$

We explain how to generate balanced random inputs in Section 4.5.

4.5 Balanced Random Inputs

We generate balanced random inputs using an algorithm that we call BRINC (Balanced Random Inputs for Neural Classifiers). The generated inputs are balanced with respect to the reference model's predictions, and therefore, can be used with the overlap metric to estimate models' similarity. Figure 4.5 shows the frequency of predicted labels for a MNIST classification model on BRINC generated random inputs. As the plot shows, the model has a perfectly balanced distribution of labels over this data. The workflow of BRINC algorithm

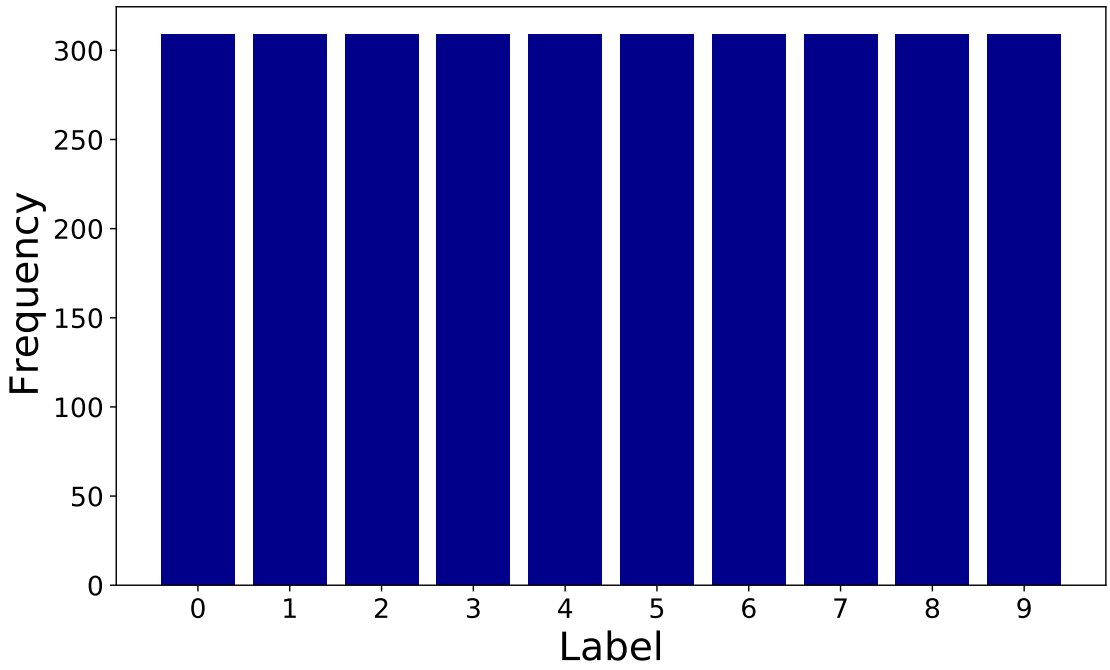


Figure 4.5: Frequency of labels with Balanced Inputs by BRINC

is inspired by the research in fuzz testing neural network models [88, 92] and is detailed in the rest of this section.

The process starts by generating a few seed inputs that cover the output labels in a balanced format. DNN fuzzing techniques [88, 92] generally start with a set of seed inputs from the model's dataset, or generate seed inputs based on some knowledge that they have regarding the valid inputs. In our case, in the absence of any knowledge about the model's inputs, not even the range of the input values, we randomly generate input vectors with values within a given range, e.g. between -1 and 1. After generating the seed inputs, at each step, an input is selected and its values are mutated by some percentage to generate new inputs with values in the same range. For example, if an input vector has 784 values, a mutation percentage of 10% results in changing 78 values. A new input is only added to the input corpus if it satisfies two conditions: (i) it adds new coverage to the network, measured by the distance that this input's prediction probability vector has with the prediction probability vectors of the inputs already in the corpus; and (ii) it maintains the balance of output labels. The first

condition is an adaptation of measuring coverage as proposed by Odena et al. [88] where distances among logits are calculated to keep track of the coverage. The second condition is specific to the goal of generating balanced random inputs and therefore, being able to use similarity thresholds that are immune to models' biases. The details of BRINC's input generation procedure are detailed in the rest of this section.

4.5.1 Algorithm

- **Seed inputs.** The process of input generation, starts with generating a few seed inputs and mutating them to generate new inputs. To this aim, starting with a set of initial inputs (we included one with zero values, and one with random float values between -1 and 1), we repeatedly mutate the seed inputs by various mutation percentages to generate inputs that cover all the output labels.

- **Next input to mutate.** At each step, selecting the next input to get mutated is done randomly from the set of inputs that predict the least frequent label at that point. Mutating an input that predicts the least frequent label, increases the chances that the mutant also predicts the same least frequent label, ensuring the balance of the input dataset.

Algorithm 5 Mutate an input

```
1: procedure MUTATE(input, range, mutPer, model)
2:     inputFlat ← getFlatShape(input)
3:     randIndexes ← randomSample(inputFlat, mutPer)
4:     for each index ∈ randIndexes do
5:         valueAtIndex ← randomFloatInRange(range)
6:         input[index] ← valueAtIndex
7:     end for
8:     return transformInputShapeToModelShape(input, model)
9: end procedure
```

- **Mutation.** Algorithm 5 shows the steps of the mutation process. First, the input is flattened (reshaped to a vector). This is to ease the process of selecting the indexes to

the values that are going to be changed, and having a uniform process for mutation of inputs at any shape. Then, $mutPer\%$ of this flattened *input*'s values are selected and changed to random float values. Parameter *range* is a tuple containing the range for generating these values. For example, the tuple (-1,1) results in generating values between -1 and 1. Finally, *input* (which now contains the mutated input) is transformed to its original shape, which is the input shape of the corresponding model, and returned.

- **Mutant acceptance.** The mutant is added to the input corpus if (i) its prediction vector's distance to its nearest neighbor (based on the euclidean distance among the prediction probability vectors) is more than a *distance* threshold, and, (ii) it predicts the least frequently predicted label (to ensure the balance of inputs). If any of these conditions is false, the mutant is discarded, and a new mutation is performed. Therefore, while we try to mutate each input in a way that the mutated version also predicts the same label, at the same time, it is ensured that the new input's prediction probability vector is far from this input's and all other inputs' prediction probability vectors. This prevents the inputs to be biased towards a specific set of prediction vectors.

- **BRINC driver and parameters.** Algorithm 6 shows BRINC's input generation procedure and its parameters. Here, *distance* is the threshold for the minimum Euclidean distance among valid mutants. *ranges* contains a set of range tuples that are used to constrain the random number generation: rather than operating on only one range (e.g. (-1,1)), the algorithm also operates on some subranges (e.g. (-1,1), (-1,0), (0,1)); empirically, we observed that certain models react better to inputs in certain subranges. $maxMut$ limits the number of mutations made in each range. $maxValid$ specifies the maximum number of valid mutants to be generated in each range. These last two parameters control the algorithm so that it changes ranges ($maxMut$) and that it eventually ends ($maxValid$). In the algorithm, the variable $generatedCount$ keeps a tally of the number of inputs generated in each range, and $noNewNum$ tracks

Algorithm 6 Generate balanced random inputs

```
1: procedure genInput(model, distance, mutPer, ranges, maxMut, maxValid)
2:     seeds ← generateSeeds(model, ranges)
3:     for each range ∈ ranges do
4:         generatedCount ← 0
5:         noNewNum ← 0
6:         while noNewNum ≤ maxMut and generatedCount ≤ maxValid do
7:             nextToMutate, leastFreqLbl ← findLeastFreqLbl(seeds)
8:             mutant ← mutate(nextToMutate, range, mutPer, model)
9:             mutantLbl ← getLabel(model, mutant)
10:            distToNearest ← findNearestNeighByPred(mutant, seeds)
11:            if distToNearest > distance and mutantLbl == leastFreqLbl then
12:                seeds.add(mutant)
13:                generatedCount ← generatedCount + 1
14:                noNewNum ← 0
15:            else
16:                noNewNum ← noNewNum + 1
17:            end if
18:        end while
19:    end for
20:    return seeds
21: end procedure
```

for how many consecutive steps in a range, no new input has been generated that satisfies the conditions. For each range, as the condition of the while loop at line 6 shows, the mutation process continues while $noNewNum$ has not passed $maxMut$ and $generatedCount \leq maxVal$.

4.5.2 Parameter Tuning

We tune the parameters of BRINC based on the number of inputs that we would like to generate and the desired diversity in predictions. We found the parameter values to be dependent on the number of labels, and in some case, the model itself. Therefore, parameter tuning for BRINC is necessary, manual or automatic.

Generally, a large *distance* results in more diverse predictions but can hamper the input generation process by preventing the mutant from covering the least frequent label. Similarly,

Table 4.1: Summary of BRINC's parameters.

Param	Definition	Suggested Value
$mutPer$	% of the *input*'s values changed at each step	5% or 10%
$distance$	Min. Euclidean distance threshold among mutants	0.001
$ranges$	Set of ranges to constrain the numbers generation	(-1,0),(0,1),(-1,1)
$maxMut$	Limit for the num. of mutations made in each range	300
$maxValid$	Max. num. of valid mutants generated in each range	500 or 1000

a large $mutPer$ results in more diverse inputs but can change the inputs drastically, hindering the least frequent label generation. A very small $mutPer$, on the other hand, may result in not satisfying the $distance$ condition. In our experiments, we found the values 5% and 10% (and in one case, 30%) to be working acceptably. Finally, a larger $maxValid$ results in generating more inputs at the expense of taking longer for the process to finish. We tried with values of 500 and 1000, depending on the number of labels (for large number of labels, larger $maxValid$ should be used since this parameter eventually defines the maximum number of rows that are generated). For the $ranges$, we chose the list $\{(-1, 0), (0, 1), (-1, 1)\}$. We experimented with other ranges as well, but did not observe major differences (even for models trained on different ranges) so we used these ranges for the sake of simplicity. We also tried different values for $maxMut$ and found the value 300 to work well.

For all input generation processes with started with parameters $distance = 0.001, mutPer = 5\%, maxMut = 300, ranges = \{(-1, 0), (0, 1), (-1, 1)\}$ and $maxValid$ of either 500 or 1000 (depending on the number of output labels), and further tuned the parameters if necessary. Therefore, we suggest starting with this set of parameters as well.

The suggested parameter values to start with are summarized in Table 4.1.

Chapter 5

Experimental Evaluation of RICA

The evaluation of RICA is done by answering four research questions. We study if random inputs, which are the primary choice of inputs in RICA, are suitable for DNNs functional similarity detection (Goal 1), and what are the trade-offs in using different similarity metrics available with RICA (Goal 2). We do this by comparing a set of models' accuracy on 5 well-known classification problems (i.e. their ground truth of functionality) against their RICA similarity with reference models. We also study if RICA is capable of detecting similarity within models for which we have no information. In addition to these two goals, we study if RICA is capable of operating on DNN models that perform a task other than classification. To this aim, we apply RICA on a regression task. Specifically, we investigate the following research questions:

- RQ1: Is RICA capable of detecting similar and dissimilar models using random inputs?
- RQ2: How do various similarity metrics used by RICA compare?
- RQ3: Can RICA detect similarities within unknown models?
- RQ4: Can RICA's effectiveness go beyond classifiers?

5.1 D56K Evaluation Dataset

This dataset consists of **56,355** classifiers collected in the following manner. We used the GitHub code search API [1] to look for files with *.h5* extension, a widely used extension for saved Keras [28] models[2], in GitHub public repositories. In total, we obtained a list of 340,933 h5 files, of which we were able to download 335,789. Next, we attempted to load these models into the Keras environment to separate DNN models from other possible objects saved in h5 files, and we identified 102,602 DNN models. After loading the models, we recorded their input and output shapes, by analyzing them using Keras. We then clustered the models based on their input and output shapes. Finally, since the focus of our work is mostly on classifiers, we filtered the clustered models based on the activation function of their last layer: we only retained models whose output layer has either a *sigmoid* or a *softmax* function. This resulted in a total of 56,355 models clustered into 6,431 groups.

5.2 Accuracy vs. Models' Similarity

Here we answer RQ1 and RQ2.

5.2.1 Methodology

We selected 5 known datasets – *MNIST, Fashion MNIST, MNIST Reverse Color, Iris [36]*, and *Sonar [40, 32]* – and trained classifiers on them to serve as our reference models. MNIST is a 10 class hand-written digit classification (black digits on white background) dataset and FMNIST is a 10 class pieces of clothing classification dataset. They both are two popular and publicly available datasets that have extensively been used in previous research [145, 90, 93].

[1] https://docs.github.com/en/rest/reference/search#search-code

[2] There are other external storage formats as well, but we limited our search to h5, because of its simplicity.

MNIST Reverse Color is a dataset curated by us by reversing the coloring of the background and foreground in the MNIST dataset. Iris is a dataset that classifies types of Iris plant into 3 classes (based on 4 features), and Sonar is a dataset that classifies sonar signals into bouncing off either Rock or Mine, based on 60 features

We then identified the clusters within our D56K dataset whose input and output shapes match, or are input compatible with, those of these models, and measured the accuracy of the models fetched from the clusters on the corresponding canonical test datasets. By using the canonical test sets, we are able to have the ground truth of the functionality of the models through their accuracy values. Empirically, we consider that when models have accuracy above 65% on the canonical test sets, they implement a good-enough classification of those inputs. When the accuracy falls below 50%, we consider the models to be classifying something different. In between 50% and 65% accuracy, we consider the models to be undecided, with just a vague functional similarity. To make sure we calculate the correct accuracy for the models, where applicable, we applied various common feature scaling techniques to the inputs, measured the accuracy of each model on all the scaled and non-scaled datasets, and considered the maximum accuracy value among these as the true accuracy of the model. In the end, 16 cases fell in the undecided region of ground-truth of functionality (between 50% and 60% accuracy on the testing dataset); we excluded those from the final calculations of the results.

Having established the ground truth of whether the models perform the classification for the chosen test sets or not, we then fed random inputs to both the reference models and these models, and measured the similarities using the three metrics described in Section 4.4.2. For generating the BRINC inputs for each reference model to be used with the overlap metric, we tuned BRINC's parameters as follows: for $ranges$ and $maxMut$ we used $\{(-1, 0), (0, 1), (1, 1)\}$, and 300 respectively. For the rest of the parameters, we started with values recommended in Section 4.5.2 and made small changes with the goal of generating at least 150 inputs per

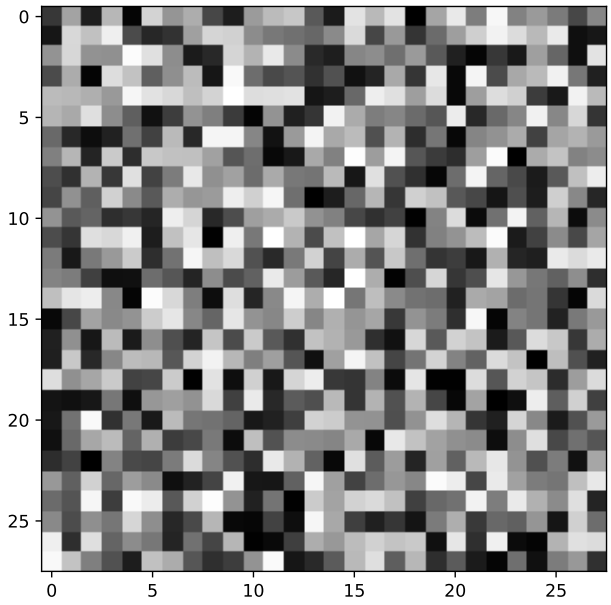


Figure 5.1: BRINC Generated Image Example

label. For the other two metrics, we generated unconstrained random inputs in range of -1 and 1. In order to show an example of what the random inputs look like in cases of image recognition tasks, Figure 5.1 shows an example of an input generated by BRINC for a model trained on the MNIST dataset.

5.2.2 Results

In total, $1,039$ comparisons were done per metric.

For each reference model and each metric, we show scatter plots that depict the relationship between the models' accuracy (x-axis), taken as the ground truth, and their similarity score with the reference model (y-axis). These plots are shown in plots depicted in Figure 5.2 through Figure 5.9. Here, "same shape" means that the reference and its candidates have the exact same input shapes, and "compatible shape" means that input shapes are not exactly the same, but compatible. The green dashed horizontal lines on each plot show the dissimilarity and similarity thresholds for the corresponding similarity metric (in case of

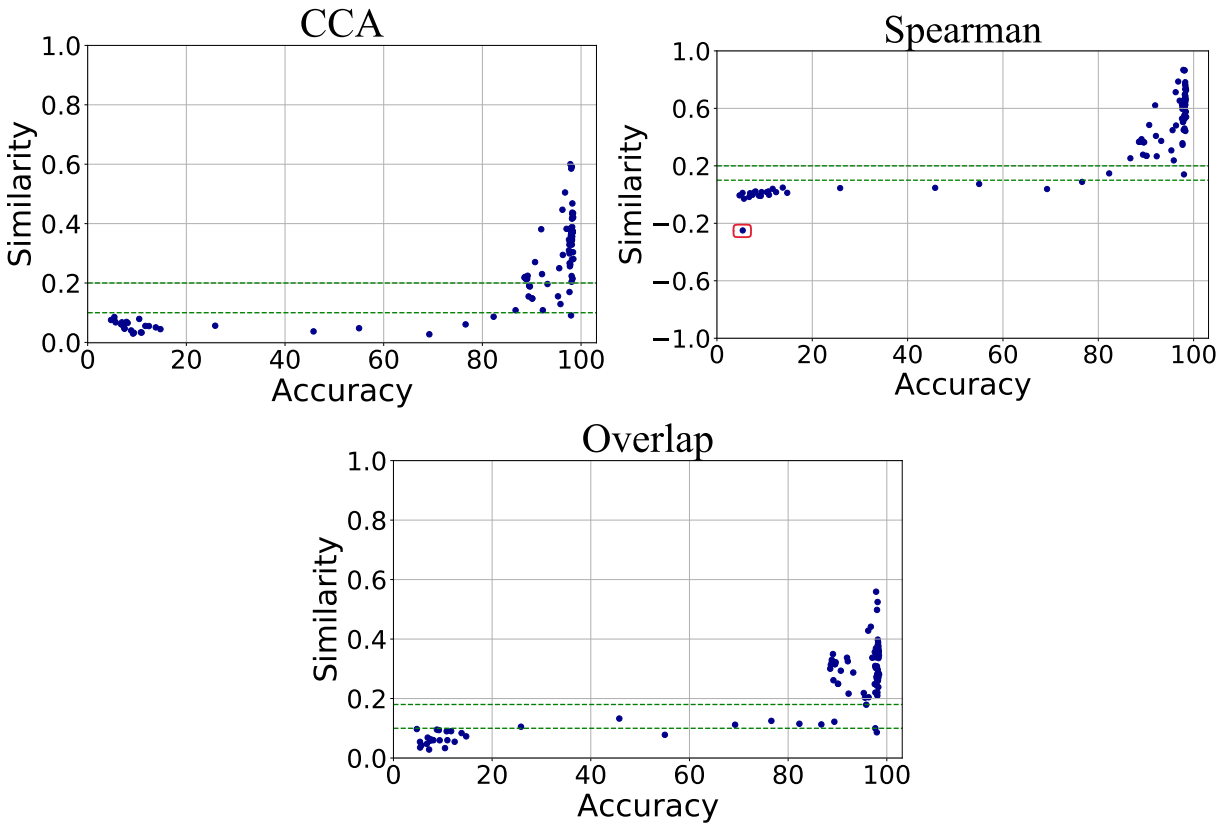


Figure 5.2: Similarity vs. Accuracy: MN Ver1 same shape as reference model

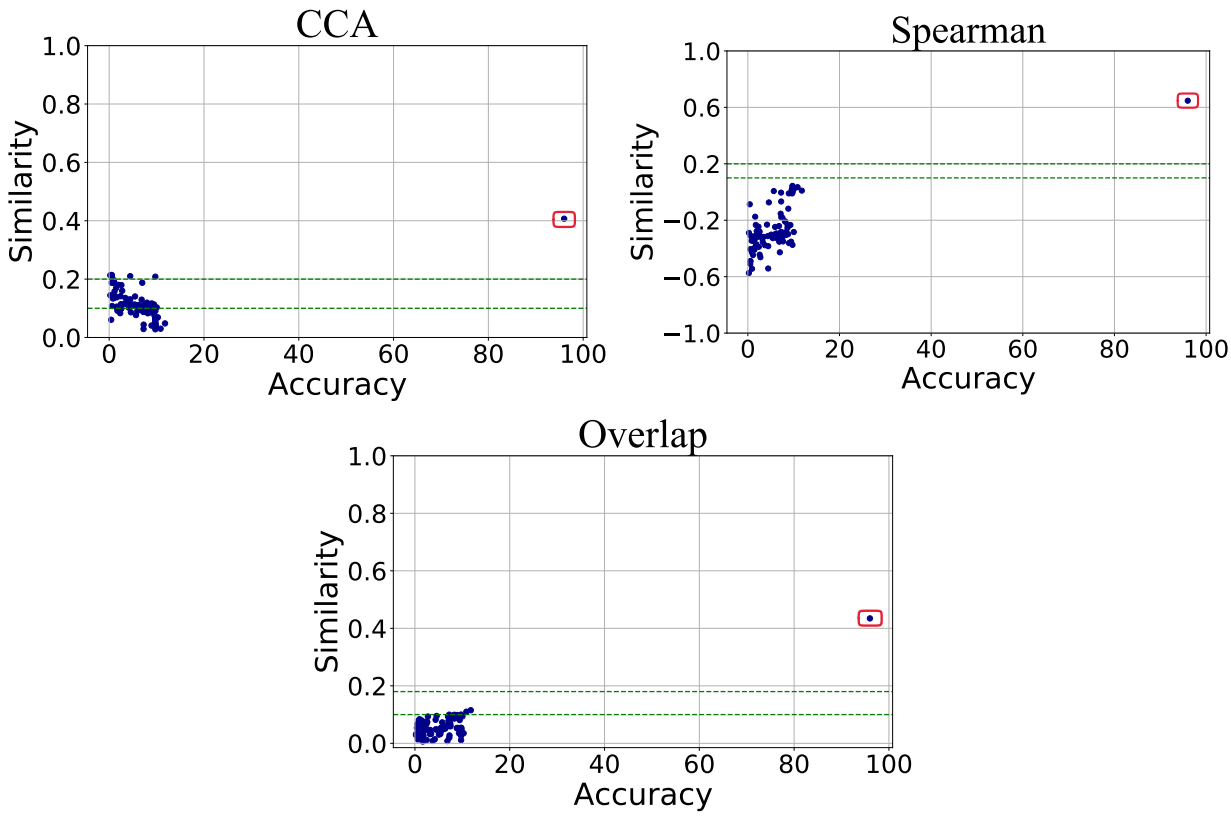


Figure 5.3: Similarity vs. Accuracy: MN Rev Clr same shape as reference model

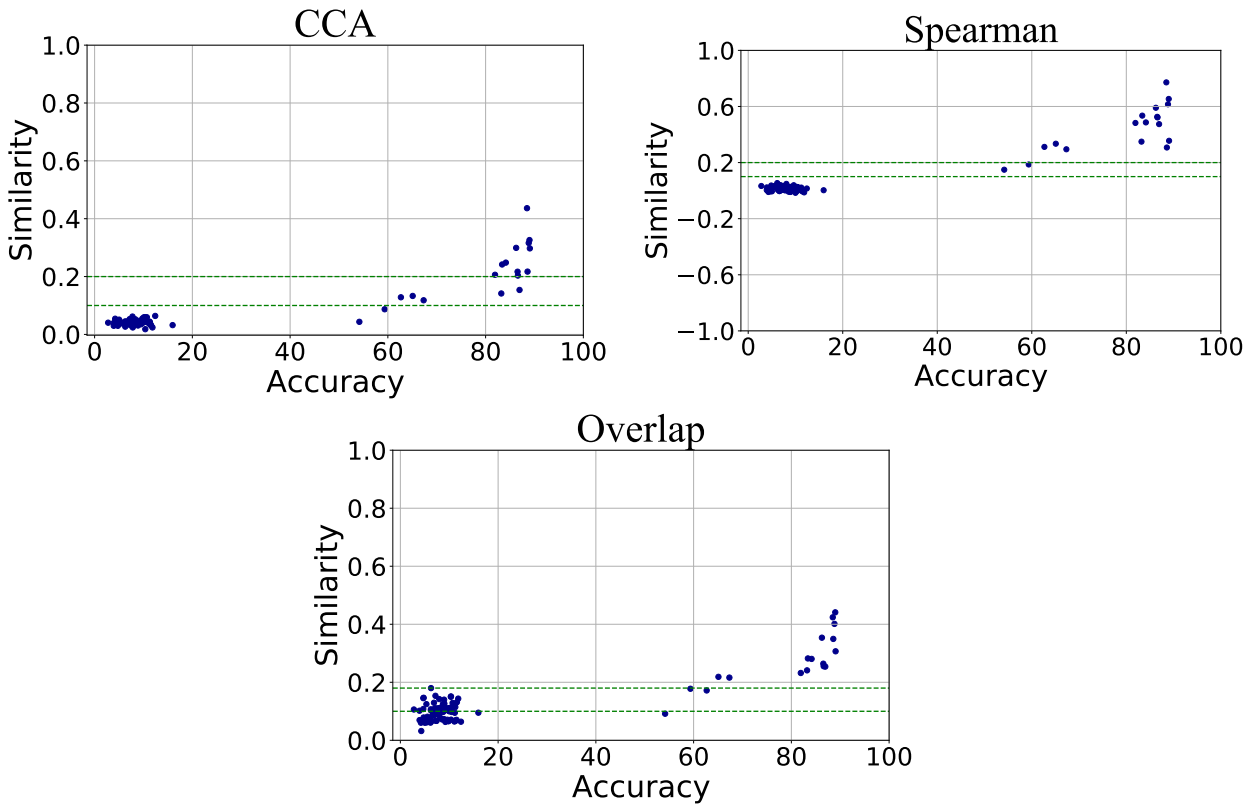


Figure 5.4: Similarity vs. Accuracy: FMN same shape as reference model

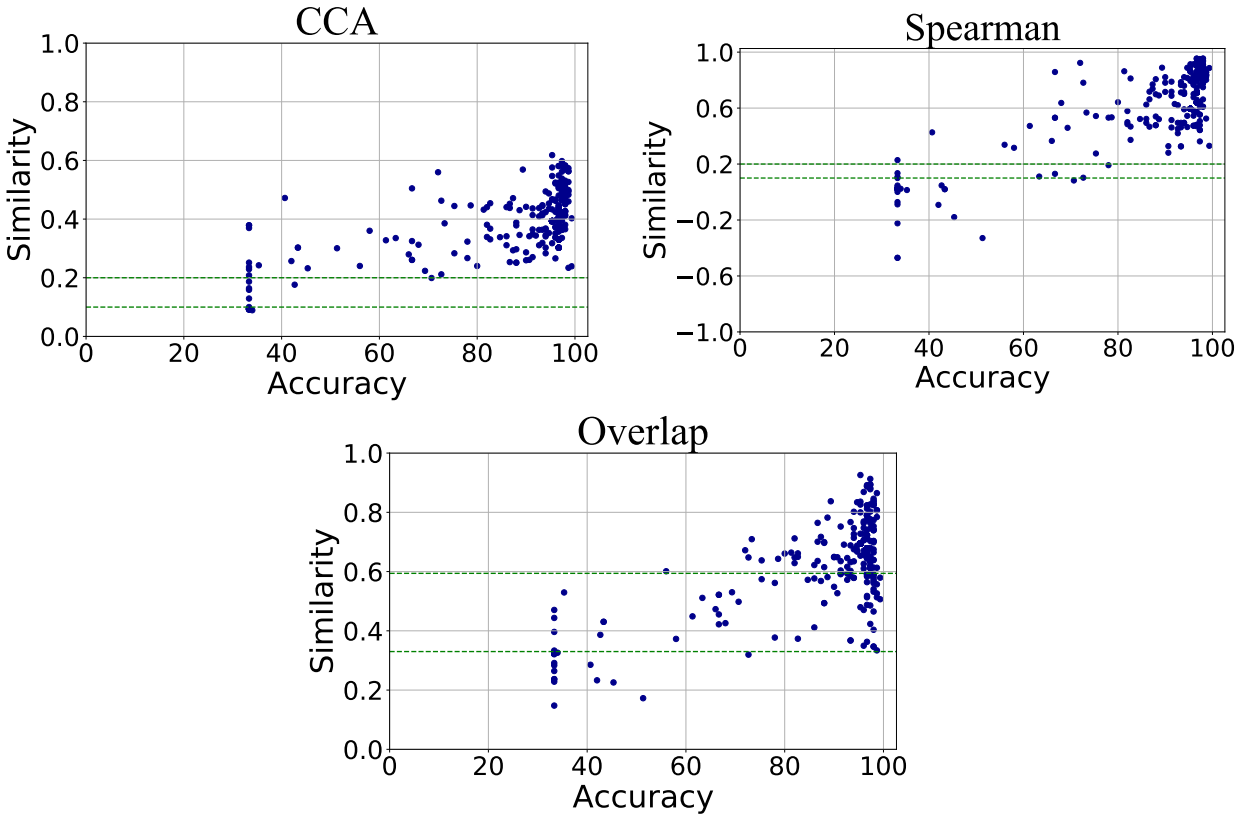


Figure 5.5: Similarity vs. Accuracy: Iris same shape as reference model

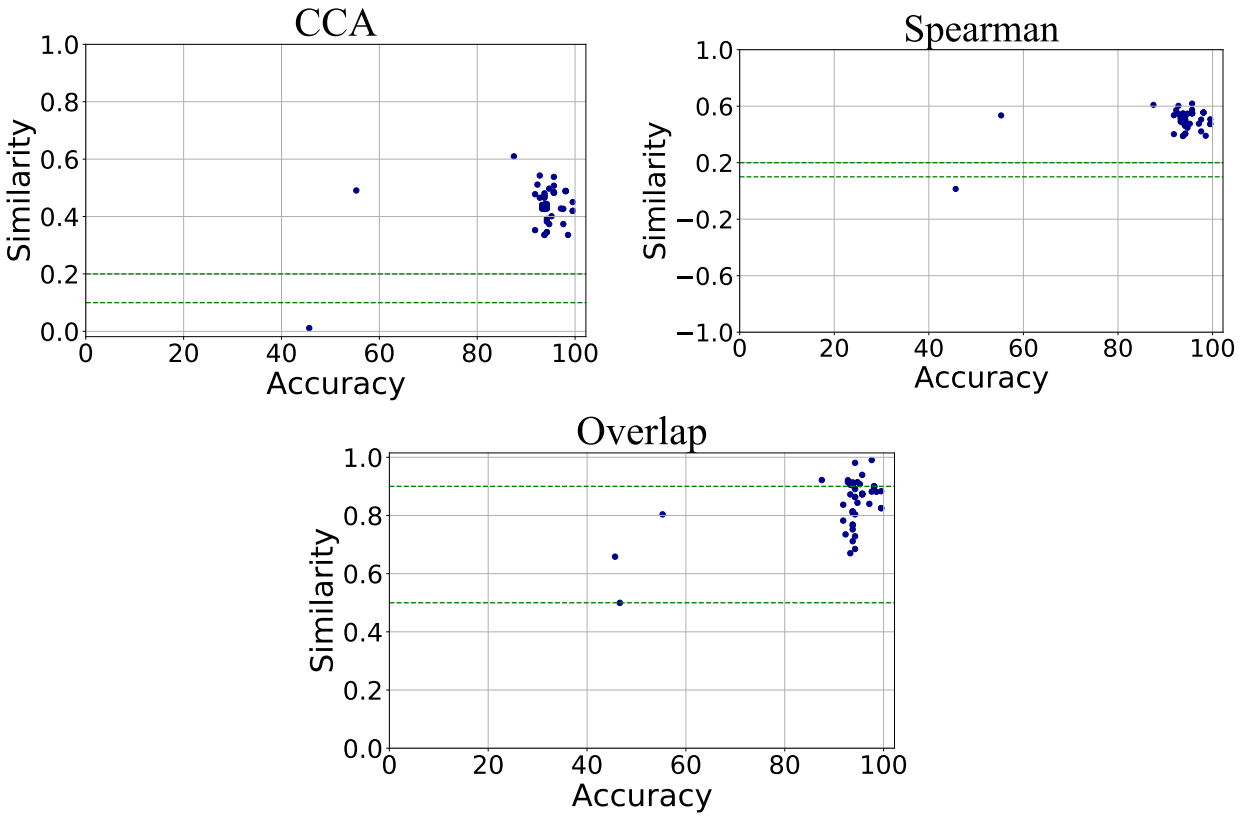


Figure 5.6: Similarity vs. Accuracy: Sonar same shape as reference model

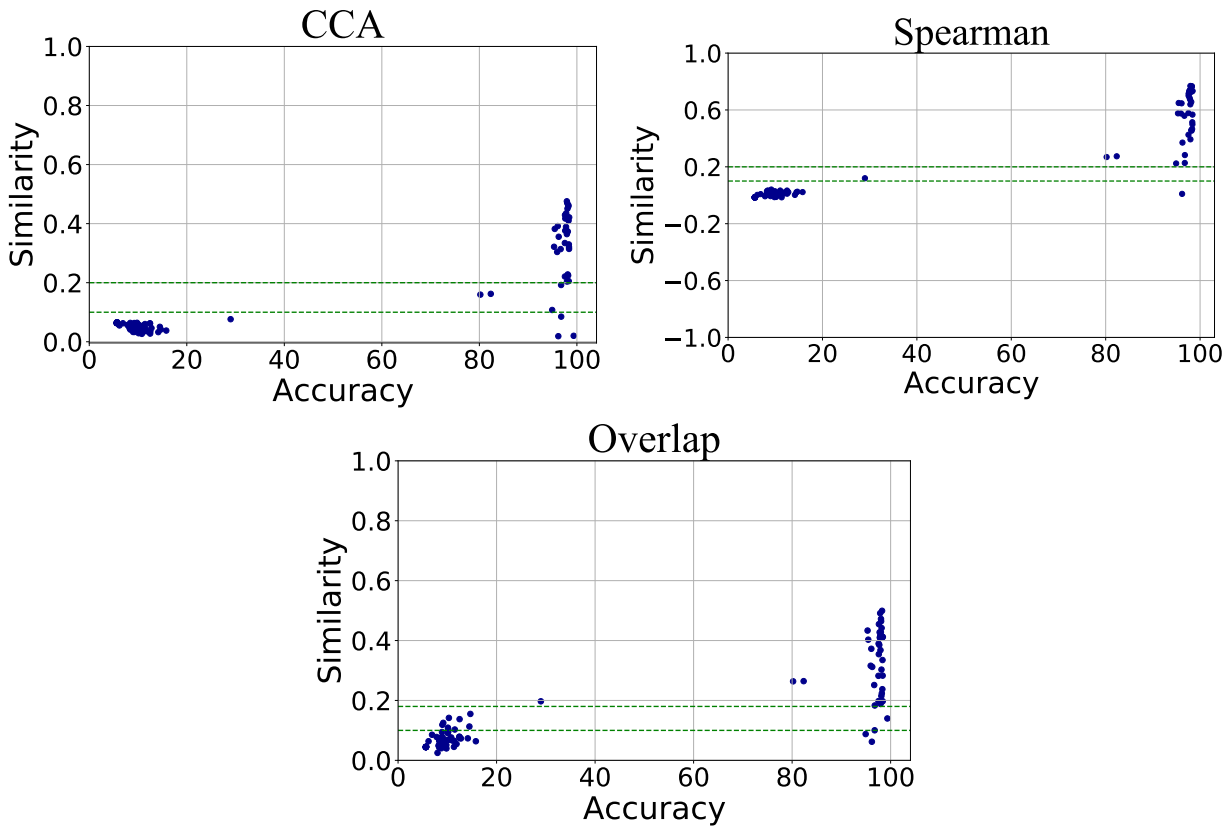


Figure 5.7: Similarity vs. Accuracy: MN Ver1 compatible shape as reference model

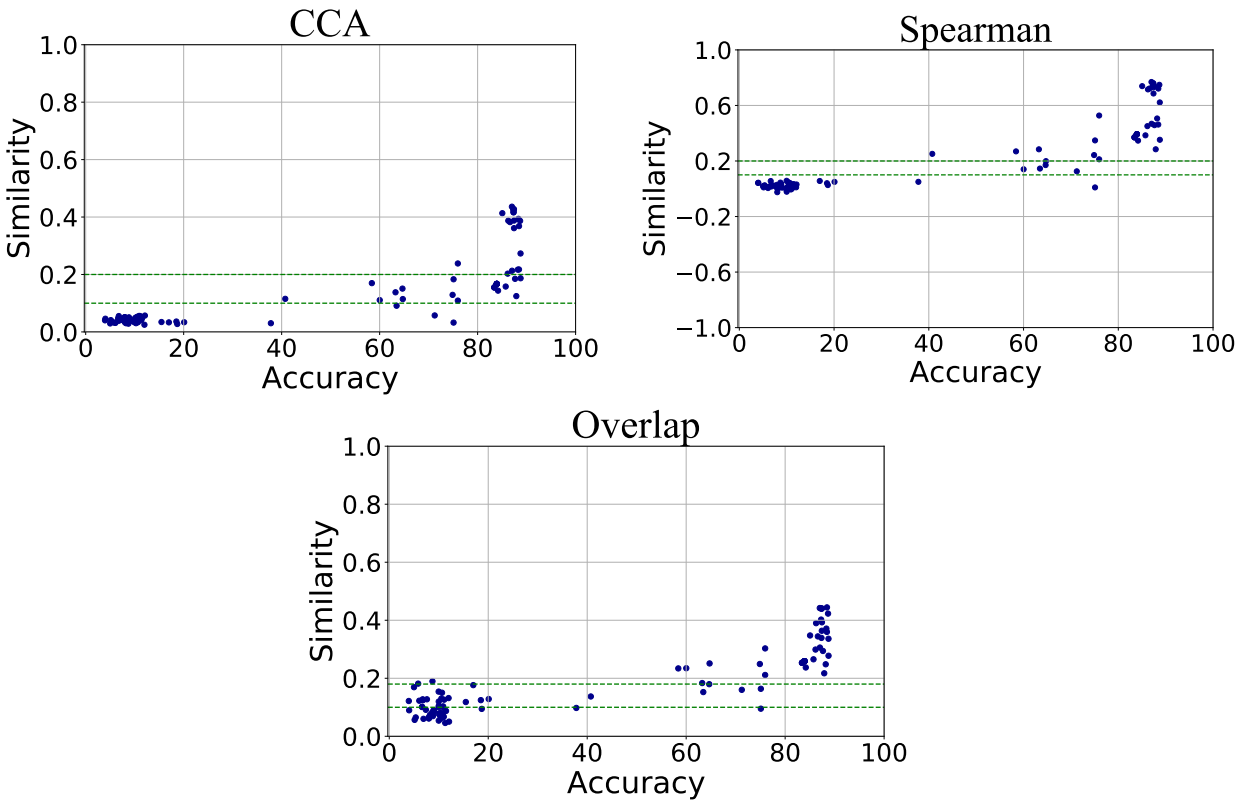


Figure 5.8: Similarity vs. Accuracy: FMN compatible shape as reference model

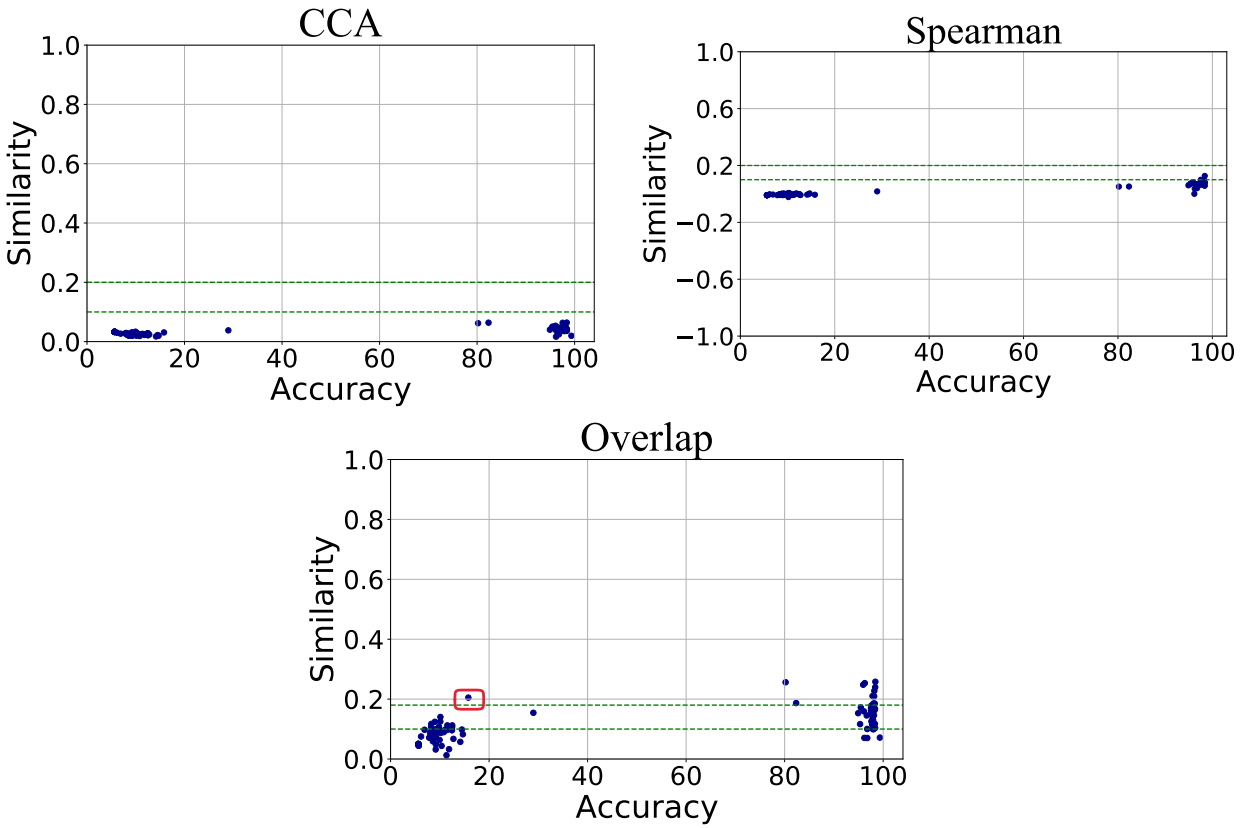


Figure 5.9: Similarity vs. Accuracy: CNN compatible shape as reference model

overlap metric, the thresholds vary based on the number of output classes n). For example, Figure 5.5 shows the accuracy vs. similarity between our trained Iris model and 376 models we found with the exact same input shape (a vector of size 4) and output shape (a vector of size 3); the accuracy values come from testing those real-world models with the canonical Iris dataset, and the similarity values for each plot come from comparing these models using the corresponding metric and random inputs. The green threshold lines are at 0.1 and 0.2 for CCA and Spearman, and at 1/3 (chance) and $2 \times 0.9/3$ (not chance) for Overlap, given 3 output classes.

Additionally, Table 5.1 quantifies RICA's successes and errors per metric for all of these comparisons. The columns below "Similarity Range" show the number of predictions that are classified as similar according to each metric's thresholds, therefore, showing the number of true positives and false positives per metric. The columns below "Dissimilarity Range" follow a same protocol for the dissimilar predictions and list the number of true negatives and false negatives per metric. The "Uncertainty Range" part shows the predictions classified as "uncertain' according to each metric's thresholds.

Furthermore, Table 5.2 shows the values of precision, recall, and accuracy using each of the metrics based on the results from Table 5.1. Here, we considered any value below the similarity threshold (including the uncertainty region) to be a negative (dissimilar) prediction.

RQ1: Is RICA capable of detecting similar and dissimilar models using random inputs? As the results show, the similarities and dissimilarities are pretty much distinguishable using all the three metrics. Visually, that can be seen in the large clusters of each plot on the right above the similarity threshold, and on the left below the dissimilarity threshold. The strong precision (over 96%), recall (over 76%), and accuracy (over 76%) numbers from Table 5.2 also attest this observation. Thus, we conclude that RICA is able to detect DNN models' functional similarity using random inputs.

Table 5.1: RICA's successes and errors per similarity metric

Reference Model	Similarity Range						Dissimilarity Range						Uncertain. Range		
	#TP			#FP			#TN			#FN			#Total		
	CCA	Spr	Ovl	CCA	Spr	Ovl	CCA	Spr	Ovl	CCA	Spr	Ovl	CCA	Spr	Ovl
MN same shp	60	72	68	0	0	0	23	23	21	4	2	2	12	2	8
MN RevClr same shp	1	1	1	6	0	0	40	99	97	0	0	0	53	0	2
FMN same shp	11	15	15	0	0	0	82	82	41	0	0	0	4	0	41
Iris	339	336	272	13	2	0	12	27	23	0	1	1	7	5	75
Sonar	60	60	25	0	0	0	1	1	1	0	0	0	1	1	36
MN compat shp	34	39	37	0	0	1	59	58	49	3	1	2	4	2	11
FMN compat shp	20	39	38	0	1	2	52	51	29	2	1	1	20	2	24
CNN compat shp	0	0	12	0	0	1	59	59	48	41	38	5	0	3	34
Total	**525**	**562**	**468**	**19**	**3**	**4**	**328**	**400**	**309**	**50**	**43**	**11**	**101**	**15**	**231**

Table 5.2: RICA's Precision, recall, accuracy per similarity metric

Precision			Recall			Accuracy		
CCA	Spr	Ovl	CCA	Spr	Ovl	CCA	Spr	Ovl
96%	**99%**	**99%**	85%	**91%**	76%	83%	**94%**	76%

RQ2: How do various similarity metrics used by RICA compare? We look at the last row of Table 5.1 to answer this question. We see that Spearman metric has the maximum number of true positive and true negative cases, and the least number of false positive and uncertain cases compared to the other two metrics. The overlap metric, on the other hand, has the least number of false negatives. These results, along with the precision, recall and accuracy numbers from Table 5.2 show that in general, Spearman is a powerful metric for detecting functional similarities among the DNN models.

Looking at the individual cases, however, we see that there are scenarios where the metrics have different behaviors. One case is the *CNN vs. compat shape*: the CCA and the Spearman metrics were not able to detect any true positives in this case, while the Overlap metric detected 12 true positive cases. A distinguishing feature of this model is that it is a convolutional neural network model, that includes convolutional layers, while none of the

models being compared to it (except for one) have a convolutional architecture. This shows that architectural differences between the models being compared may pose challenges for the similarity metrics used by RICA and can prevent them from making accurate predictions. This issue is something that we will fully investigate in Section 6.1. For now, however, the results show that the Overlap metric, is still able to detect a few true positive cases in such scenarios, as opposed to the other two metrics that predicted almost all cases to be dissimilar.

These results show that in general, Spearman is a suitable metric for detecting models' similarity. In certain situations, for example where there is substantial differences between models' architectures (which can be checked given the model file), the overlap metric (used with BRINC random inputs) is a more suitable choice. When compared with Spearman, CCA is not showing superior results. It, however, has a better recall and accuracy compared to the Overlap metric, while Overlap has a better precision.

We also discuss some interesting cases that we observed during these comparisons.

MN Rev Clr vs. same shape: One interesting observation here pertains to the Spearman metric that has identified MN Rev Clr to have an inverse relationship with several models (negative correlation values), reflecting the inverse coloring of the training datasets. The other observation is one model with accuracy $> 90\%$ on MN Rev Color's dataset and a high similarity value with this model using all three metrics. This case is circled in all scatter plots of Figure 5.3. Since MN Rev Color's training data was curated by us, we did not expect to find any similar models. So we investigated this case's GitHub repository and its training code, and it turned out that the same transformation that we applied to the MNIST dataset is applied on this model's dataset. This model is also circled in the Spearman scatter plot of Figure 5.2: the Spearman metric calculated a similarity of ≈ -0.25 between this model and the MN Ver1 reference model, showing a moderate inverse relationship between these two.

CNN vs. compatible shape: The odd case in this group is a model with very low accuracy (less than 20%) that scored above the similarity threshold only with the overlap metric, circled in the Overlap scatter plot of Figure 5.9. This could be evidence of a Type-1 error (false positive), so we analyzed this model's GitHub repository: the analysis revealed that this is a case of a model that performs digit recognition with digits similar to what exist in the MNIST dataset, but trained on a different dataset created by the author of that project. As such, the overlap metric with the use of BRINC generated input was able to flag it as similar to our CNN model. The other two metrics did not detect this case.

5.3 Unknown Models

Here we answer RQ3 about the possibility of finding similarities among unknown models using RICA. To this aim, we performed a set of experiments to find the similarity scores among the 56,355 classification models that we collected from GitHub (D56K dataset). Details of these experiments are described below.

5.3.1 Overall Similarity Results

Methodology

The analysis was done in an intra-cluster manner, meaning that all models inside a cluster (each cluster includes models with the same input and output shapes) were paired together and their similarity with each other was computed. This limited the scope of the experiments to only comparing the models that have the same input and output shapes, and therefore, helping in managing the scale of the experiments. As a result of this, clusters that only included one model were skipped. It should be noted that models not in the same cluster

can also be candidates of similarity, but those cases were not considered here to manage the scope of the experiments.

The primary similarity metric used in these experiments was Spearman since as attested by the presented experiment results, it is the most precise metric among the metrics used with RICA. In cases were Spearman correlation produced a NaN value (cases where the standard deviation of one of the prediction probability vectors of the models being analyzed is zero for all given inputs), we used CCA as the similarity metric. If CCA also returned NaN, then such cases were discarded.

Results

Through the course of these experiments, there were models for which we could not get their predictions (it either took a long time to get their predictions or there were errors with models). Such cases were removed from the experiments. In total, in the end, $7,368,065$ comparisons were done. After removing the rows with NaN similarity values, $6,963,916$ rows were left. Among these, $6,696,213$ similarity values were computed with the Spearman metric and the rest of them $(267,703)$ were computed with CCA. Table 5.3 shows a statistical overview of these comparisons. As the table shows, 26% of the comparisons resulted in detecting similar models, 63% detected dissimilar models, and only 11% of the comparisons resulted in cases falling into the uncertain region of similarity. These results show that in 89% of cases RICA was able to make a decision about the models' similarity with high certainty, which is very promising result, attesting the applicability of this method in real world scenarios when no information about the models being analyzed is available.

Figure 5.10 shows the distribution of the calculated similarity values. The two dotted vertical lines show the dissimilarity and similarity threshold lines. As the plot in this figure shows, most of the similarity values lie between 0 and 0.25. We also found cases where the similarity

Table 5.3: Similarity Statistics for GitHub Models

Similarity Status	Number	Percentage
Similar	1,815,671	26%
Not similar	4,393,832	63%
Uncertain	754,413	11%
Total	6,963,916	100%

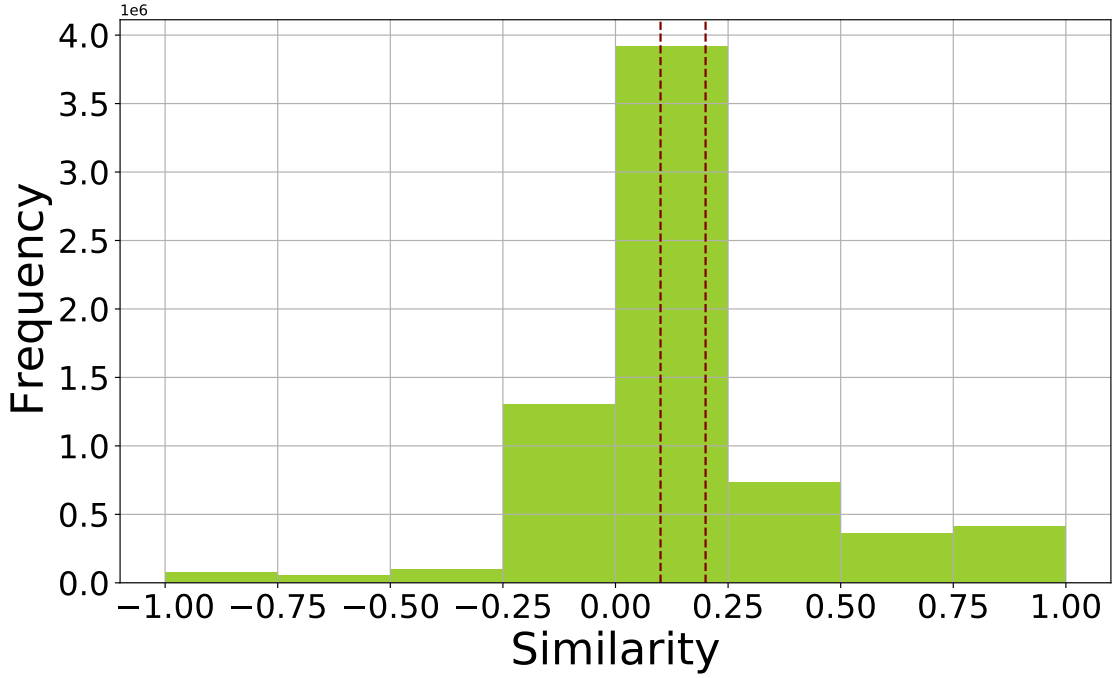


Figure 5.10: Similarity values for GitHub models

values were equal to one (100% similarity) attributing to models that are duplicates of each other in terms of their ultimate functionality. In total, $67,548$ of such exact clone cases where found which accounts for $\approx 0.01\%$ of total comparisons (after removing NaN values).

5.3.2 Manual Analysis

To further understand how RICA works in detecting similarities among the models that we have no or little knowledge about and dive deeper into the reported similarity values, we randomly selected two clusters from our D56K dataset and one classifier from each cluster as

the reference model. We then calculated similarities between each reference model and other models in the cluster, using the two metrics that have the highest precision: Spearman and Overlap. For one reference model, a raw dataset file with string and categorical variables was found in its GitHub repository along with code transforming these feature values to numerical values appropriate for the model. Therefore, although a dataset is available, using it needs unknown processing steps and human effort that are hard to automate. In the case of the other reference model, we did not find any related dataset files in the repository. Figure 5.11 shows the similarity values calculated (x-axis) and their frequencies (y-axis) for the model with Raw data available, per metric. The dashed vertical lines show each metric's dissimilarity and similarity thresholds. Figure 5.12 shows the same information for the model with no available data. Parameter tuning for BRINC (for generating inputs suitable for the Overlap metric) was done similar to Section 5.2.

Raw Data Available: The cluster includes 34 models, where the input is a vector of 26 and the output is binary. The reference model's repository is aimed at "telecommunication customer churn detection". Based on Figure 5.11, Spearman detected 7 similar cases. From these, the Overlap metric detected one as similar, and placed the rest in the uncertainty region, two with a high similarity (over 0.8) and the other four with ≈ 0.61 similarity. For the three models with highest similarities by both metrics, their repositories indicate churn prediction, hence, we considered them to be true similar cases. The repositories for the other four models describe them as aimed at people's distress recognition or analysis, and therefore, we did not consider them to be true positives reported by Spearman.

No Data Available: The cluster has 56 models with input shape of $48 \times 48 \times 1$ and output shape of 6. The reference model's repository describes it as a "facial emotion recognition" project. As the plots in Figure 5.12 show, both metrics detected five exact matches (100% similarity): in three cases, the repository's description mentions face or emotion recognition, and in two cases, no information is given. The Overlap metric detected other similar cases

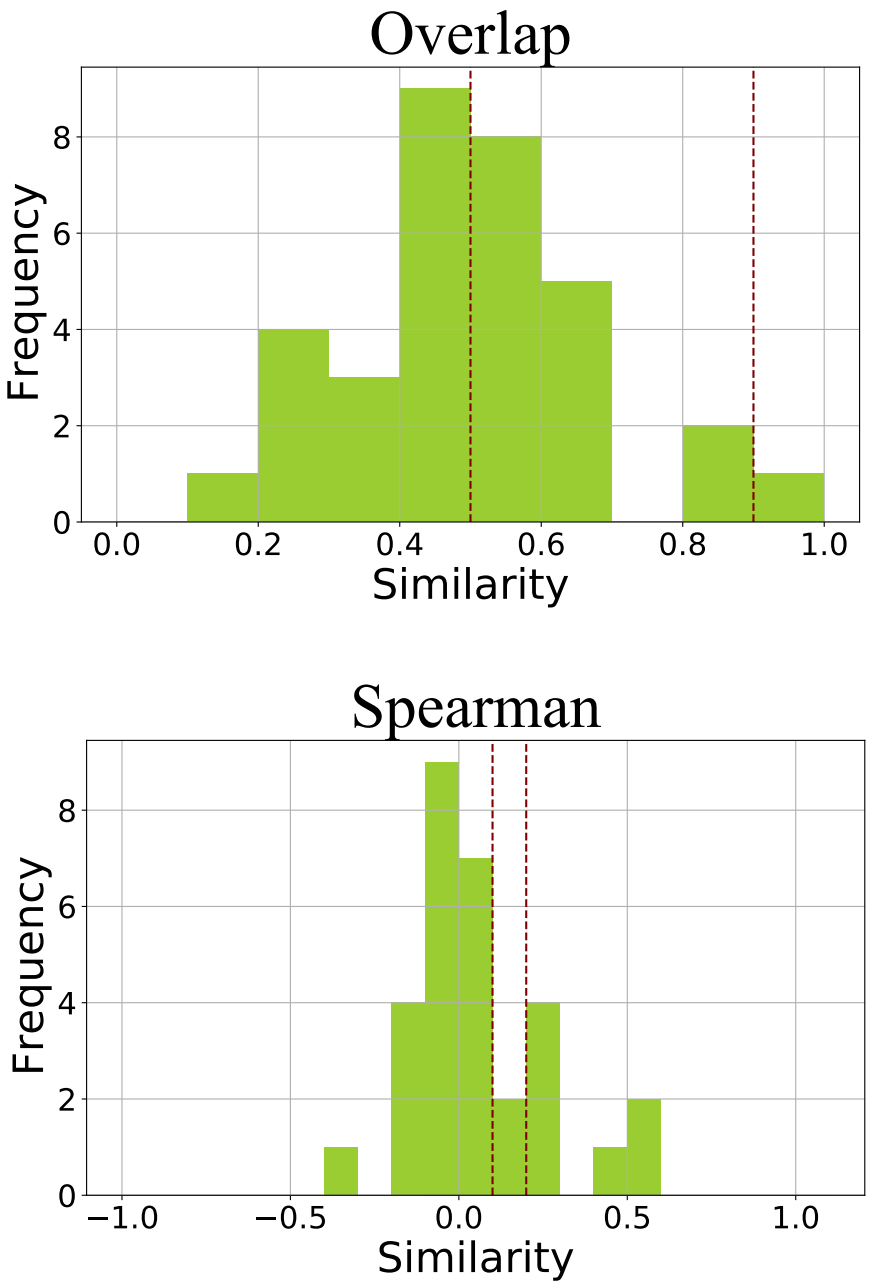


Figure 5.11: Similarity values for Raw Data Model

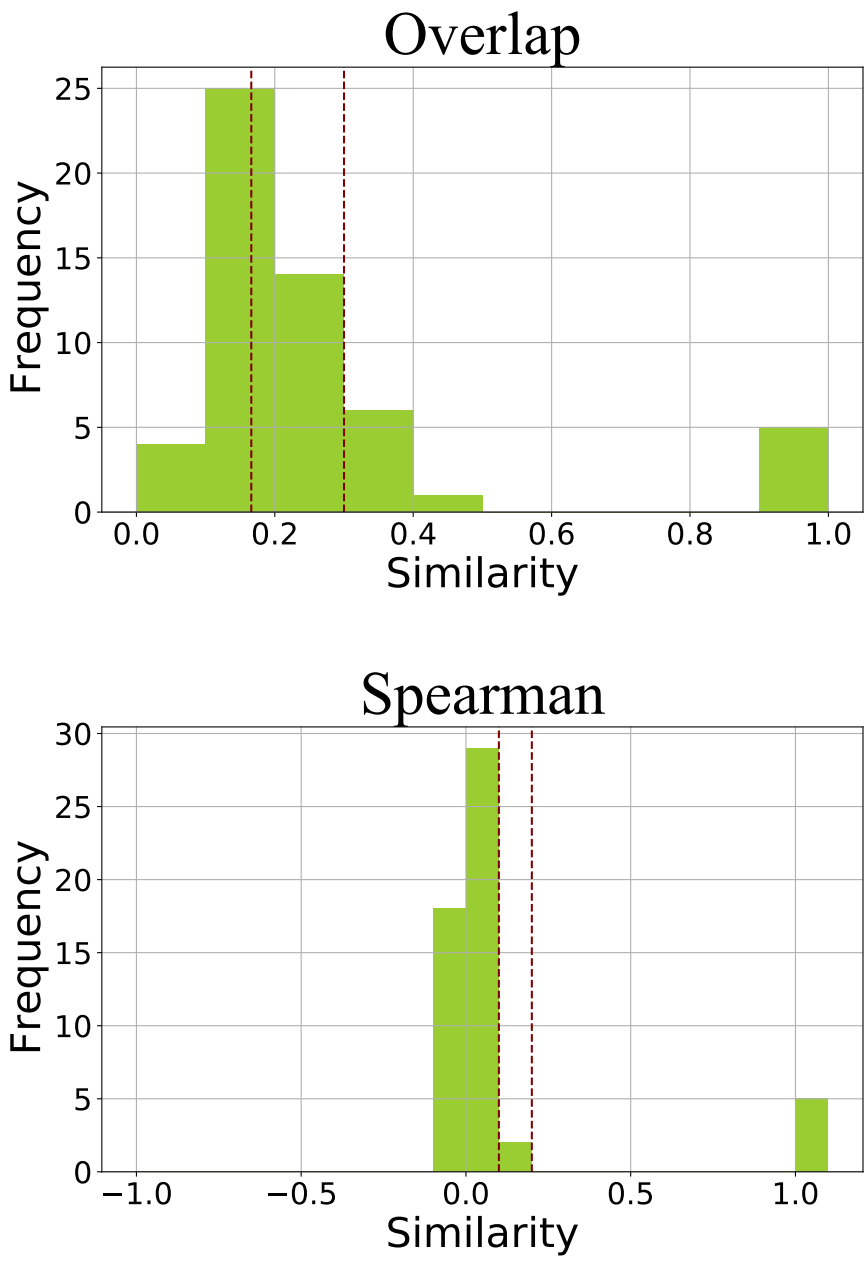


Figure 5.12: Similarity values for No Data Model

too: one with ≈ 0.42 similarity where the repository's description is emotion recognition and six cases with similarities > 0.3. In three cases, the repository mentions either face or emotion recognition, and in the rest, no description was found. These cases were not detected as similar by the Spearman metric.

Based on the results from the experiments conducted on unknown models in this section, we can answer **RQ3: Can RICA detect similarities within unknown models?** As attested by the results, RICA is able to detect similarities among the models for which we do not have any (or have little) information, with the use of random inputs.

5.3.3 Discussion

As the results presented in this section show, random inputs are a viable replacement of canonical test inputs for purposes of similarity detection of neural classifiers. The three metrics studied and used by RICA are capable of detecting similarity/dissimilarity quite accurately. Spearman correlation shows the best results in most experiments. Spearman is also easy to use, as it needs simple unconstrained random inputs, and produces results bound to the range of [-1,+1] with well established dissimilarity/similarity thresholds. This metric is also able to detect inverse relationships between the models. CCA is also easy to apply, and it is similar to Spearman correlation in terms of inputs and thresholds, and produces similarity results in the range of [0,1]. The overlap metric is more complicated to apply, as it requires the generation of balanced random inputs. However, this metric was shown to perform better than the others in the comparison of models with substantial architectural differences.

All in all, as the results show, the three metrics complement each other; there are cases where one metric detects similarity and the others report values that fall within uncertainty ranges. Therefore, for the most accurate results, using a combination of the metrics is recommended,

and the final conclusion can be derived based on the results of all.

5.4 Similarity Analysis of Regression Models

As the experiments presented in this chapter demonstrated, RICA is very effective in finding the functional clones of DNN classifiers. Since the two metrics of Spearman and CCA used with RICA are generic enough to be applied to other types of DNN models as well, we ran another series of experiments to measure the effectiveness of RICA for other types of DNNs. We selected a regression problem for this purpose and since Spearman metric was shown to be the most precise metric, we experimented with this metric. The results of these experiments, which are presented in this section, help us answer RQ4 about the capability of RICA on working beyond classifiers.

5.4.1 Dataset

We used the Boston housing prices dataset [43] to test RICA on a regression problem. This is a problem to predict housing prices in Boston based on a set of 13 features. The target price variable in this dataset ranges from 5 to 50. We loaded this dataset from the Keras library.

5.4.2 Experiments

Methodology

To perform the experiments, we trained a feed forward fully-connected network on the Boston housing prices dataset to serve as our reference model. We then queried the initial models

dataset that we downloaded from GitHub (before filtering out the models to only keep the classifiers for creating the D56K dataset) for models that fit the input and output shapes of this dataset, and we were able to find 88 models to serve as our candidate models.

Similar to the experiments presented for the classification models, we need to establish a ground truth of similarity to compare the similarity predictions with. The accuracy metric is not applicable here since the target y values are continuous integer values, and accuracy is best fitted for problems where the target value is of categorical format. A suitable metric for regression problems is R^2 which measures the amount of variability in the dependent variable that can be explained by the model [118]. In other words, it explains the quality of predictions by measuring the amount of variability observed in them. To this aim, it divides the sum of the squared of the prediction error by the total sum of the square where the calculated prediction is replaced with mean [118]. R^2 is calculated using the following formula.

$$R^2 = 1 - \frac{\sum_i (y_i - \hat{y}_i)^2}{\sum_i (y_i - \bar{y}_i)^2} \tag{5.1}$$

R^2 values typically range from 0 to 1 with values closer to 1 showing a better fit between the predicted and the actual value. It can, however, assume negative values as well if the model fits the data very poorly. We calculated R^2 for all the candidate models using the testing portion of the Boston housing prices data and used this value as the ground truth of similarity. In calculation of R^2, similar to previous experiments, we applied various common feature scaling and normalizations to the data and calculated R^2 using all of the resulting datasets, and used the maximum value among all these as the final value. In cases where the R^2 became negative, we substituted the calculated value with zero for easing the visualization and interpretation of results.

Results

Figure 5.13 shows the similarity values calculated between the reference model and all the candidate models with RICA using the Spearman metric (y-axis) versus the ground truth R^2 values (x-axis). As the plot shows, in majority of cases, the similarity values agree with the calculated R^2 values. For larger similarity values, we see that the majority of R^2 values are more than 0.6, and for similarity values less than 0.1 (the threshold of dissimilarity), the R^2 values are mostly zero.

There are, however, two cases with similarity values higher than 0.2 (the threshold of similarity) where the calculated R^2 values are zero. In one case, the calculated similarity is ≈ 0.76 and in the other one, it is ≈ 0.37. We investigated both of these cases by looking at their GitHub repositories. In the case of the model with similarity value ≈ 0.76, the GitHub repository showed that the model is aimed at predicting Boston housing prices; however, the data was loaded from an external source and the separation of training and testing sets was done through a randomized process of shuffling and selecting row indexes to be included in training/testing sets. In addition, the ranges of numbers (both for features and the target y value) were different from our dataset and all its rescaled versions. Therefore, in this case, we concluded that RICA correctly predicted the model to be similar to our reference model (as they both are aimed at the Boston housing prices problem), but due to differences between the training data used for this model and our reference model, the R^2 value is not representative of the ground truth of similarity. This is similar to the case of the CNN model aimed at MNIST classification which was discussed in Section 5.2.

In the case of the other model in Figure 5.13 with similarity value ≈ 0.37, we were not able to find any information from the model's GitHub repository that could explain the low R^2 value.

The results presented here us answer the following research question: **RQ4: Can RICA's**

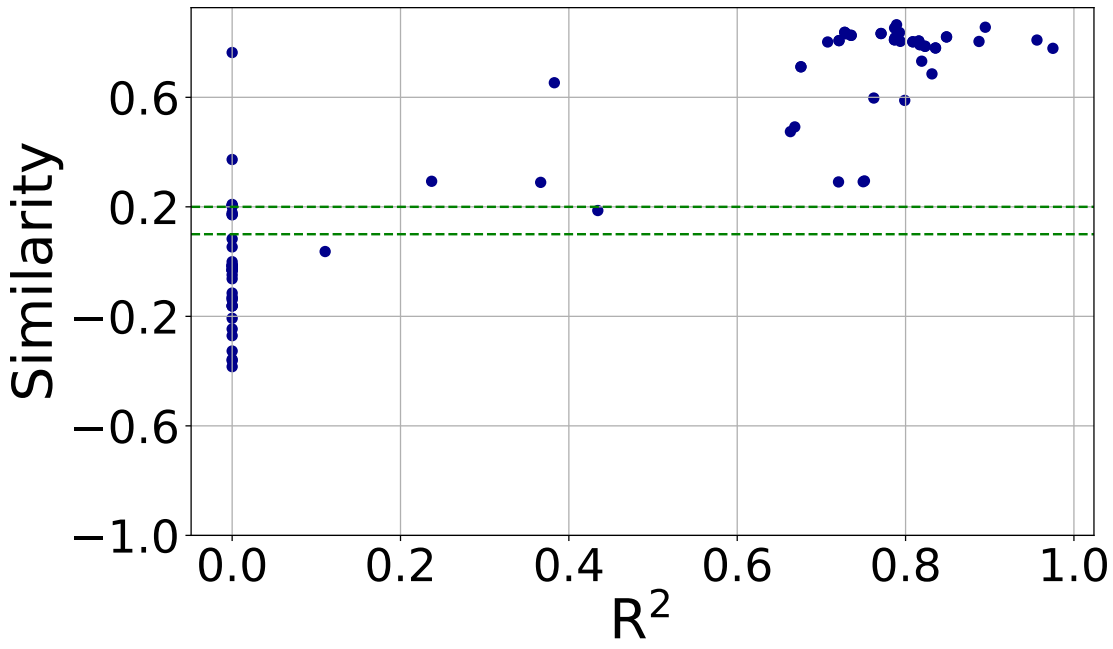


Figure 5.13: Similarity values vs. R^2 for Boston housing prices models

effectiveness go beyond classifiers? The results demonstrate that in addition to classification which was extensively studied in this dissertation, RICA has the potential to be applied to other tasks targeted by DNN models. Specifically, we presented experiments that confirm its applicability in regression problems.

5.5 Threats to Validity

In all the experiments with the classification models, we assumed that the ordering of output labels is the same for the reference and the candidate models. Although it is unlikely to have different orderings, specifically for well-known datasets, the existence of such cases can affect our results, both the calculated accuracy of the models and the similarity predictions.

Chapter 6

Analytical Discussions on RICA

In this chapter, I present a set of experiments aimed at diving deeper into the capabilities of RICA.

In Section 6.1, I detail a set of sensitivity analysis experiments whose goal is to understand how the randomized nature of inputs and the models' various properties, which are mostly related to their training, affect RICA's similarity results. As such, the following research questions will be investigated:

- RQ5: What is the effect of RICA's input generation randomization on models' response to similarity detection?
- RQ6: What is the effect of the models' characteristics (training randomization, training parameters, accuracy, architectures, training datasets) on their response to RICA?

Next, I present a taxonomy for DNN clone types that we derived based on our observations from the experiments that we did with RICA, and discuss what clone types RICA is capable of detecting. These clone type categories are inspired by the clone types presented in the literature for code clones, which were discussed in Chapter 3 of this dissertation.

6.1 Sensitivity Analysis

6.1.1 Sensitivity Analysis Dataset (D14)

We carry out a set of sensitivity analysis experiments where we study models' functional similarity with respect to various independent variables related to models' training and input randomization, to answer RQ5 and RQ6. For these experiments, we trained 14 models using MNIST (10 class hand-written black digit classification on white background) and FMNIST (10 class pieces of clothing classification) datasets. These models are presented in Table 6.1, and are grouped in clusters in a way that the models in each cluster have one varying property with respect to the first model, MN Ver1, which we consider to be the reference model. The group "Instances" consists of two identical models in all aspects, except for being trained separately, therefore being subjected to the randomness of the training process. The next group, "Train Params", consists of models with different training parameters as to the reference model, the "Architectures" group includes models that have architectural differences with respect to the reference model, and the "Datasets" group consists of models that have been trained on different datasets, with varying levels of similarity with the training data of the reference model. The column "Acc" shows the accuracy of models on their own test data, and the column "X-Acc" is the accuracy of the models on the reference model's test data. X-Acc, therefore, serves as the ground truth for similarity of the models with respect to the reference model. Therefore, MN Rev Color (which has a reverse coloring of background and foreground as to the reference model) and FMN are considered to be dissimilar to the reference model as they both have a very low X-acc. All of these models classify 28×28 gray-scale images, reshaped to vectors of 784, into one of 10 possible classes.

Table 6.1: Sensitivity analysis: trained models (D14 dataset)

Group	Name	Description	Acc	X-Acc
Instances	MN Ver1 (Ref)	2 layers fully connected with ReLU in 1st layer, #epochs=10, learning rate =0.001, MNIST training data scaled to (0,1)	98.16%	98.16%
	MN Ver2	Same architecture, training params, and training data as Ref model, trained separately	98.15%	98.15%
Train Params	MN Long Tr	Same architecture, training data, and learning rate as Ref model, trained for 50 epochs	98.35%	98.35%
	MN LR01	Same architecture, training data, and epochs as Ref model, learning rate=0.01	97.80%	97.80%
	MN LR18	Same architecture, training data, and epochs as Ref model, learning rate=0.18	75.92%	75.92%
Architectures	MN Sig	2 layers fully connected with Sigmoid in 1st layer, training parameters and training data same as Ref model	97.68%	97.68%
	MN Deep NN	4 layer fully connected with ReLU in the 1st 3 layers, training parameters and training data same as Ref model	98.21%	98.21%
	MN CNN	CNN with 2 convolution, 2 pooling, and 2 fully-connected layers, training params and training data similar to Ref model	99.18%	99.18%
	MN CNN2	Same architecture, training params, training data as MN CNN, trained separately	99.06%	99.06%
Datasets	MN 1st Batch	Same architecture and training params as Ref model, trained on MNIST training data's 1st 30K rows in the scale of (0,1)	97.55%	97.55%
	MN 2nd Batch	Same architecture and training params as Ref model, trained on MNIST training data's 2nd 30K rows in the scale of (0,1)	98.39%	98.39%
	MN No-scale	Same architecture and training params as Ref model, trained on MNIST training data in range of (0,255)	97.21%	90.74%
	MN Rev Color	Same Architecture and training params as Ref model, trained on MNIST black background & white digit scaled to (0,1)	95.27%	1.67%
	FMN Model	Same architecture and training params as Ref model, trained on Fashion MNIST training data scaled to (0,1)	88.88%	8.35%

6.1.2 Sensitivity Analysis Results

In this section, I present a set of experiments using Dataset D14 to answer RQ5 and RQ6. To answer these research questions, we present Figure 6.1 and Table 6.2. Figure 6.1 shows the box&whiskers plot of the similarity (for each metric) between the reference model (MN Ver1) and all the other models of dataset D14. Each box corresponds to similarity values computed using 10 different random input datasets (unconstrained inputs for CCA and Spearman, and BRINC inputs for Overlap). The parameter values to generate BRINC inputs were: $distance = 0.001, mutPer = 5\%, ranges = \{(-1, 0), (0, 1), (-1, 1)\}, maxMut = 300, maxValid = 1000$.

Table 6.2 shows the results of additional per metric similarity measurements when taking each of the models under study as the reference model. Black cells are 100% similarity; grey cells are values above the similarity threshold of each metric, white cells are values below the dissimilarity threshold, and red cells fall in the uncertainty zone between these two thresholds. Except for the values below "MN Ver1" column, which corresponds to the medians in Figure 6.1, all numbers are single-run experiments.[1] In explaining our findings, we will refer to both Figure 6.1 and Table 6.2.

RQ5: What is the effect of RICA's input generation randomization on the models' response to similarity detection? We look at the variations of similarity results when computed using 10 separately generated random inputs to answer this question. Figure 6.1 shows that the effect of randomization is negligible for CCA and Spearman metrics. The boxes corresponding to these two metrics are small with minimum, maximum, and median being almost the same. For Overlap metric, we see some variability within the boxes, which is, however, bound to relatively narrow bands. That can be seen in the values of the 1^{st} and 3^{rd} quartiles, as well as in the min and max values for each box. In particular, we

[1] Some of the red cells may escape the uncertainty zone with further measurements.

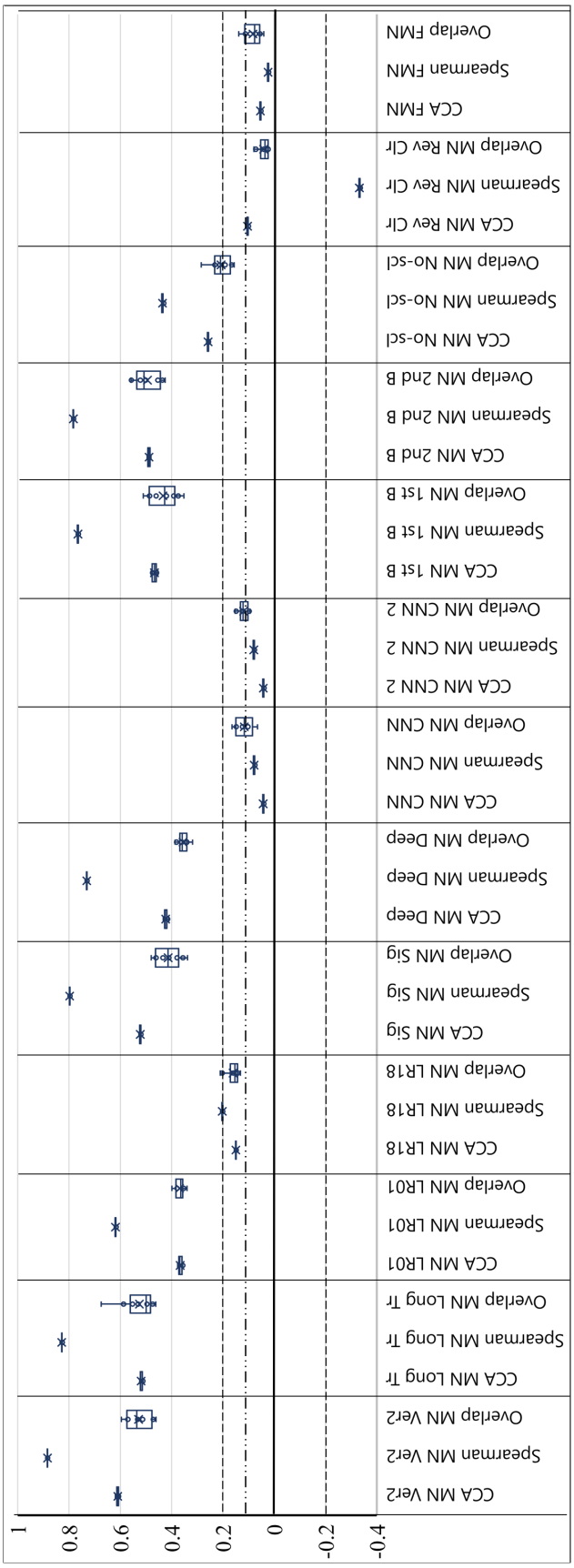


Figure 6.1.: Similarity with MN Ver1. Each box: minimum, 1^{st} quartile, median, 3^{rd} quartile, maximum.

Table 6.2: Similarity between all models under study using the three metrics.

	Reference model																																				
	Instances						Train Params									Architectures										Datasets												
Candidate	MN Ver1			MN Ver2			MN Long Tr			MN LR01			MN LR18			MN Sig			MN Deep			MN CNN			MN CNN2			MN 1st B			MN 2nd B			MN No-scl			MN Rev Clr			FMN		
↓	CCA	Spr	Ovl	CCA	Spr	Ovl	CCA	Spr	Ovl	CCA	Spr	Ovl	CCA	Spr	Ovl	CCA	Spr	Ovl	CCA	Spr	Ovl	CCA	Spr	Ovl	CCA	Spr	Ovl	CCA	Spr	Ovl	CCA	Spr	Ovl	CCA	Spr	Ovl	CCA	Spr	Ovl	CCA	Spr	Ovl
MN Ver1				0.61	0.88	0.47	0.52	0.83	0.30	0.36	0.62	0.18	0.15	0.20	0.13	0.52	0.80	0.41	0.43	0.73	0.23	0.04	0.08	0.11	0.04	0.08	0.09	0.46	0.77	0.36	0.49	0.78	0.38	0.26	0.44	0.31	0.11	-0.33	0.08	0.05	0.02	0.12
MN Ver2	0.61	0.88	0.54				0.51	0.82	0.37	0.35	0.61	0.17	0.14	0.20	0.17	0.50	0.79	0.43	0.41	0.72	0.29	0.04	0.07	0.16	0.04	0.08	0.09	0.45	0.76	0.47	0.47	0.78	0.46	0.26	0.43	0.32	0.11	-0.33	0.06	0.05	0.02	0.11
MN Long Tr	0.52	0.83	0.50	0.51	0.82	0.53				0.35	0.60	0.18	0.13	0.19	0.18	0.41	0.71	0.44	0.36	0.66	0.36	0.04	0.08	0.11	0.04	0.09	0.08	0.37	0.69	0.48	0.40	0.71	0.38	0.25	0.44	0.35	0.10	-0.32	0.03	0.04	0.02	0.09
MN LR01	0.37	0.62	0.36	0.36	0.61	0.37	0.35	0.60	0.26				0.19	0.25	0.17	0.33	0.57	0.37	0.34	0.57	0.27	0.05	0.09	0.21	0.05	0.09	0.11	0.31	0.55	0.29	0.34	0.57	0.45	0.28	0.46	0.37	0.13	-0.37	0.06	0.04	0.04	0.12
MN LR18	0.15	0.20	0.15	0.14	0.20	0.13	0.13	0.19	0.18	0.19	0.25	0.14				0.15	0.21	0.21	0.14	0.20	0.08	0.04	0.03	0.18	0.04	0.04	0.15	0.14	0.20	0.20	0.16	0.20	0.16	0.16	0.18	0.35	0.12	-0.20	0.07	0.02	0.00	0.08
MN Sig	0.52	0.80	0.41	0.50	0.79	0.34	0.41	0.71	0.28	0.33	0.57	0.19	0.15	0.21	0.15				0.37	0.63	0.18	0.04	0.06	0.26	0.04	0.06	0.12	0.45	0.75	0.39	0.49	0.77	0.40	0.25	0.44	0.39	0.12	-0.40	0.05	0.06	0.03	0.06
MN Deep	0.42	0.73	0.36	0.41	0.72	0.30	0.36	0.66	0.28	0.34	0.57	0.21	0.14	0.20	0.15	0.37	0.63	0.32				0.04	0.06	0.10	0.04	0.07	0.09	0.32	0.61	0.37	0.35	0.64	0.27	0.22	0.37	0.27	0.10	-0.26	0.03	0.04	0.00	0.11
MN CNN	0.04	0.08	0.12	0.04	0.07	0.08	0.04	0.08	0.14	0.05	0.09	0.10	0.04	0.03	0.09	0.04	0.06	0.15	0.04	0.06	0.11				0.29	0.49	0.29	0.04	0.07	0.10	0.04	0.07	0.17	0.07	0.09	0.12	0.06	-0.08	0.08	0.02	0.00	0.11
MN CNN2	0.04	0.08	0.12	0.04	0.08	0.08	0.04	0.09	0.11	0.05	0.09	0.13	0.04	0.04	0.08	0.04	0.06	0.12	0.04	0.07	0.14	0.29	0.49	0.29				0.04	0.07	0.10	0.05	0.07	0.15	0.07	0.10	0.12	0.06	-0.08	0.05	0.02	0.00	0.07
MN 1st B	0.47	0.77	0.43	0.46	0.76	0.39	0.38	0.69	0.32	0.31	0.55	0.18	0.15	0.20	0.12	0.46	0.75	0.40	0.32	0.61	0.25	0.04	0.07	0.15	0.04	0.07	0.12				0.30	0.59	0.44	0.25	0.41	0.32	0.11	-0.34	0.06	0.05	0.01	0.13
MN 2nd B	0.49	0.78	0.51	0.48	0.78	0.45	0.40	0.71	0.29	0.34	0.57	0.23	0.16	0.20	0.20	0.49	0.77	0.35	0.35	0.64	0.29	0.04	0.07	0.09	0.05	0.07	0.10	0.30	0.59	0.38				0.25	0.41	0.35	0.11	-0.34	0.04	0.05	0.02	0.10
MN No-scl	0.26	0.44	0.21	0.26	0.43	0.15	0.25	0.44	0.13	0.28	0.46	0.15	0.16	0.18	0.07	0.25	0.44	0.19	0.22	0.37	0.17	0.07	0.09	0.10	0.07	0.10	0.10	0.25	0.41	0.14	0.25	0.41	0.23				0.19	-0.38	0.01	0.04	0.02	0.18
MN Rev Clr	0.10	-0.33	0.04	0.11	-0.33	0.02	0.10	-0.32	0.02	0.13	-0.37	0.08	0.12	-0.20	0.03	0.12	-0.40	0.08	0.10	-0.26	0.04	0.06	-0.08	0.08	0.06	-0.08	0.05	0.11	-0.34	0.04	0.11	-0.34	0.04	0.19	-0.38	0.02				0.04	0.00	0.12
FMN	0.05	0.02	0.08	0.05	0.02	0.11	0.04	0.02	0.05	0.04	0.04	0.11	0.02	0.00	0.02	0.06	0.03	0.11	0.04	0.00	0.11	0.02	0.00	0.09	0.02	0.00	0.10	0.05	0.01	0.07	0.05	0.02	0.09	0.04	0.02	0.13	0.04	0.00	0.05			

notice that the variability is lower for lower values of similarity, and that, for the higher values of similarity, the minimum is very far from the thresholds. This is a good finding from an engineering point of view, as it shows that one single run of input generation will often be enough to determine models' similarity. When the first measurement falls close to the thresholds, it may be necessary to repeat the measurements and take the median as the best approximation.

RQ6: What is the effect of the models' characteristics (training randomization, accuracy, architectures, training datasets) on their response to RICA?

Training randomization: We look at the "Instances" group, MN Ver1 and MN Ver2, two virtually identical models with virtually identical accuracies to investigate this. As Figure 6.1 shows, Spearman and Overlap predicted the highest median of similarity for these two models, and CCA predicted one of the highest, as we expected. Therefore, training randomization does not show to have affected the results. Additional evidence of the low sensitivity of this independent variable can be seen in Table 6.2 for MN CNN and MN CNN2: the similarity between them and the other models is low, but between the two of them it is well above the similarity threshold for all three metrics.

Training parameters: Looking at the models in "Train Params" group in Figure 6.1, we see that increasing the number of epochs (MN Long Tr) doesn't seem to have a strong effect on the measured similarity. The min, median, 1^{st} quartile, and 3^{rd} quartile are very close to the those observed for MN Ver2 for all three metrics. The learning rate (MN LR01 and MN LR18), however, seems to have a measurable effect. For MN LR01, the similarity values in are still well above the similarity threshold, but for MN LR18, the numbers are much lower. The reason may be that increasing the learning rate makes the models converge faster and this may hurt their generalization abilities.

Models' accuracy: Most of the studied models have accuracies $> 95\%$, with two excep-

tions: MN LR18 (≈ 76% accuracy), and FMN (≈ 89% accuracy). Figure 6.1 shows that the median value of MN LR18's similarity with the reference model is either close (Spearman) or inside (CCA and Overlap) the uncertainty region. For FMN model that has a better accuracy, we see that the median values are correctly below the dissimilarity threshold. Looking at column FMN in Table 6.2, we see that CCA and Spearman have always predicted the expected similarity results for this model and overlap has uncertain results in some cases, which are, however, mostly close to the dissimilarity threshold. This means that the predicted similarity values can be sensitive to the accuracy of the models if the accuracy is sufficiently low. The reason can be related to the fact that low accuracies hurt models' prediction capabilities.

Architectures: The Architectures group includes two models (MN Sig and MN Deep) with minor architectural differences with the reference model, and two models (MN CNN and MN CNN2) with substantially different architectures with the reference model, as they include convolutional layers. Figure 6.1 shows that for the models with minor differences (MN Sig and MN Deep), the medians of similarity are well above the similarity threshold. The similarity values for CNN models, however, are barely above the chance threshold for the overlap metric, and always in the dissimilarity region for the other two metrics. When the CNN models are used as queries (columns MN CNN and MN CNN2 in Table 6.2), similarity predictions are still mostly wrong, except between the two CNN instances and some cases of overlap metric. This shows that similarity detection with RICA using random inputs can fail to compute accurate similarity measurements when the architectures are substantially different. The Overlap metric might be able to detect some similar cases in these situations.

Training datasets: Looking at the Datasets group in Figure 6.1 (MN 1st B onward), we see that similarity calculations are relatively robust with respect to functionality-preserving training data changes (MN 1st B, MN 2nd B, MN No-scl), and are correct in classifying as dissimilar the models trained on data that produces functionally different classifiers (MN

Rev Clr and FMN). This can also be seen in Table 6.2: for the first three models of the Datasets group, the cells are most gray (except for when comparisons are made with CNN models), and for the the last two models, the cells are mostly white.

6.2 Taxonomy of DNN Clones

In developing source code clone detectors, a widely accepted taxonomy of clones [103, 123] has been quite helpful. Neural network models are different from human-written code, and do not fit that taxonomy. However, the study of similarity of models can benefit from the classification of types of neural clones. Here, we suggest a taxonomy that helps us explain our targets and limitations.

Type-CC: Type-CC clones are models whose outputs agree 100% on any and all compatible inputs. This scenario happens when the models' networks, weights and biases are exactly the same, or differ only in dead neurons. In other words, "carbon copies."

Type-S: Type-S clones are structural clones, where the architecture of the models is similar, but the models can perform different functions due to significant differences in training datasets. Type-S clones can be checked by inspecting the structure of the networks.

Type-SF: Type-SF clones have similar architectures (Type-S) and also perform similar functions. With black boxes such as DNNs, Type-SF clone detection requires an input dataset for testing functions.

Type-F: Type-F clones perform similar functions, but have considerably different architectures – different number of layers, different activation functions, different types of layers, etc.

Type-S detection is trivial; the target of our work is the detection of the other types of

clones. RICA (with its use of three studied metrics along with random inputs) was shown to perform well for Type-CC and Type-SF. It was also shown to be capable of detecting Type-F clones in cases when the architectural differences are not substantial. As the architectural differences become significant, it becomes harder for RICA to detect similar functionality, and it starts reporting false negatives, as we observed in comparing fully-connected neural network models with CNN models. Further research is necessary in the realm of Type-F clones.

Chapter 7

Conclusions

7.1 Dissertation Summary

There has been an increasing amount of interest in software clone detection in recent years, for the many applications that clone detection brings. Recent work in this area has focused on proposing novel techniques that can detect clones that have no or low syntactic similarity but are semantically similar. Such clones are often called functional clones.

On the other hand, we are also observing a continuous growth in AI and machine learning areas. This unprecedented advancement in artificial intelligence has resulted in the integration of machine learning, and majorly, neural network models in different kinds of software, ranging from simple mobile apps to more sophisticated systems such as self-driving cars. As neural network models are more and more being integrated in software systems, and subsequently, specifying various functions of these systems, the need has arisen for functional clone detection techniques to consider these models in their analyses.

In this dissertation, I first reviewed previous work in the area of code clone detection and

DNN similarity measurement. I then briefly reviewed Oreo, a tool to find clones in the twilight zone, which is the area of clone types where syntactic similarity decreases while semantic (or functional) similarity is preserved. I also presented a study on the precision of code clone detection tools which highlighted the impact of considering various clone types when doing precision measurements. This study revealed the significance of performing precision studies on Type III to Type IV clone types.

I also formulated and discussed the problem of finding functional similarities among neural network models in the absence of canonical inputs, and proposed a method named RICA for this purpose. RICA generates random inputs to be used in lieu of canonical inputs for the purpose of similarity detection, and measures models' similarity by analyzing their outputs on this input. I extensively studied three similarity metrics, namely Spearman, CCA, and Overlap, that can be used with RICA for the purpose of quantifying the similarity of models based on their outputs on random inputs. To the best of our knowledge, this is the first large-scale study aimed at this problem. I also introduced a method, named BRINC, for generating balanced random inputs for classifiers to be used with the Overlap metric.

I presented several experiments that showed the effectiveness of RICA. As a part of these experiments, I curated a dataset of over 56K DNN classifiers from GitHub and clustered them based on their input and output shapes. Results of the experiments performed using this dataset demonstrated that similarity detection of DNN models using random inputs is possible, and that all three studied metrics can provide promising results, the Spearman metric, however, stood out, both in terms of ease of use and the overall results. Furthermore, I presented additional experiments to show how RICA can be applied to problems other than classification. To this aim, I applied RICA on a regression problem. As the results of all the conducted experiments, RICA was shown to be an effective method for DNN models clone detection. Finally, I presented a sensitivity analysis and a taxonomy of clone types that highlighted RICA's scope and limitations.

7.2 Future Work

DNN functional clone detection is a novel research area that can be studied in numerous ways. The work presented here, specifically, can be extended by applying the approach followed by RICA on other types of DNN models and investigate any adaptations that might be needed in those scenarios. These adaptations can range from different ways of generating inputs to using other similarity metrics. Another line of future work is to investigate how RICA can be improved to increase its accuracy for cases where there is substantial architectural differences between the models being analyzed. This would ease the detection of Type-F neural clones. Possible solutions range from other ways of generating inputs to using other suitable similarity metrics.

As mentioned earlier, one use-case of RICA is in model search. An interesting line of future work is to build a model search tool using RICA and study the effectiveness of this tool in various scenarios, such as transfer learning. Furthermore, going forward, DNN model clone detection will need to be combined with conventional clone detection tools. This will expand the applicability of these tools so that they can be applied on AI-enabled software.

Bibliography

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

[2] Inter-rater agreement. http://core.ecu.edu/psyc/wuenschk/docs30/InterRater.pdf. Accessed: 2018-10-13.

[3] Calculate similarity — the most relevant metrics in a nutshell. https://towardsdatascience.com/calculate-similarity-the-most-relevant-metrics-in-a-nutshell-9a43564f533e, 2019. Accessed: 2022-03-07.

[4] Incydr™ scoop: Data exposure jumps as employees head for the doors. https://www.code42.com/blog/incydr-scoop-data-exposure-jumps-as-employees-head-for-the-doors//, 2020. Accessed: 2022-03-02.

[5] Machine learning attack series: Stealing a model file. https://embracethered.com/blog/posts/2020/husky-ai-machine-learning-model-stealing/, 2020. Accessed: 2022-01-10.

[6] 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.

[7] Ambient Software Evolution Group. IJaDataset 2.0. http://secold.org/projects/seclone, January 2013.

[8] L. An, O. Mlouki, F. Khomh, and G. Antoniol. Stack overflow: A code laundering platform? In *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 283–293, Feb 2017.

[9] 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.

[10] 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.

[11] A. Arpteg, B. Brinne, L. Crnkovic-Friis, and J. Bosch. Software engineering challenges of deep learning. In *2018 44th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, pages 50–59. IEEE, 2018.

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

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

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

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

[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] G. T. Becker, D. Strobel, C. Paar, and W. Burleson. Detecting software theft in embedded systems: A side-channel approach. *IEEE Transactions on Information Forensics and Security*, 7(4):1144–1154, 2012.

[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. Berger. *Geometry I*. Springer, 1987.

[22] M. Bojarski, D. Del Testa, D. Dworakowski, B. Firner, B. Flepp, P. Goyal, L. D. Jackel, M. Monfort, U. Muller, J. Zhang, et al. End to end learning for self-driving cars. *arXiv preprint arXiv:1604.07316*, 2016.

[23] R. Brixtel, M. Fontaine, B. Lesner, C. Bazin, and R. Robbes. Language-independent clone detection applied to plagiarism detection. In *2010 10th IEEE Working Conference on Source Code Analysis and Manipulation*, pages 77–86. IEEE, 2010.

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

[25] J. Chen, M. H. Alalfi, T. R. Dean, and Y. Zou. Detecting android malware using clone detection. *Journal of Computer Science and Technology*, 30(5):942–956, 2015.

[26] 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*, pages 175–186. ACM, 2014.

[27] 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.

[28] F. Chollet et al. Keras. `https://keras.io`, 2015.

[29] 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.

[30] L. Deng, G. Hinton, and B. Kingsbury. New types of deep neural network learning for speech recognition and related applications: An overview. In *2013 IEEE international conference on acoustics, speech and signal processing*, pages 8599–8603. IEEE, 2013.

[31] 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.

[32] D. Dua and C. Graff. UCI machine learning repository, 2017.

[33] S. Ducasse, M. Rieger, and S. Demeyer. A language independent approach for detecting duplicated code. In *Proceedings of the IEEE International Conference on Software Maintenance (ICSM'99)*, pages 109–118. IEEE, 1999.

[34] 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.

[35] F. Farmahinifarahani, V. Saini, D. Yang, H. Sajnani, and C. V. Lopes. On precision of code clone detection tools. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 84–94. IEEE, 2019.

[36] R. Fisher. Iris. UCI Machine Learning Repository, 1988.

[37] M. Gabel, L. Jiang, and Z. Su. Scalable detection of semantic clones. In *Proceedings of the ACM/IEEE 30th International Conference on Software Engineering (ICSE'08)*, pages 321–330. IEEE, 2008.

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

[39] 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.

[40] R. P. Gorman and T. J. Sejnowski. Analysis of hidden units in a layered network trained to classify sonar targets. *Neural networks*, 1(1):75–89, 1988.

[41] 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.

[42] F. Harel-Canada, L. Wang, M. A. Gulzar, Q. Gu, and M. Kim. Is neuron coverage a meaningful measure for testing deep neural networks? In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 851–862, 2020.

[43] D. Harrison Jr and D. L. Rubinfeld. Hedonic housing prices and the demand for clean air. *Journal of environmental economics and management*, 5(1):81–102, 1978.

[44] 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.

[45] F. Hermans, B. Sedee, M. Pinzger, and A. Van Deursen. Data clone detection and visualization in spreadsheets. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 292–301. IEEE, 2013.

[46] H. Hotelling. Relations between two sets of variates. *Biometrika*, 28:321–337, 1936.

[47] B. Hummel, E. Juergens, L. Heinemann, and M. Conradt. Index-based code clone detection: incremental, distributed, scalable. In *IEEE International Conference on Software Maintenance (ICSM 2010)*, pages 1–9. IEEE, 2010.

[48] 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.

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

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

[51] L. Jiang, G. Misherghi, Z. Su, and S. Glondu. Deckard: Scalable and accurate tree-based detection of code clones. In *Proceedings of the 29th International Conference on Software Engineering*, pages 96–105. IEEE Computer Society, 2007.

[52] 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.

[53] 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*, ISSTA '09, pages 81–92, New York, NY, USA, 2009. ACM.

[54] 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.

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

[56] J. H. Johnson. Substring matching for clone detection and change tracking. In *Proceedings of the International Conference on Software Maintenance (ICSM)*, volume 94, pages 120–126, 1994.

[57] E. Juergens, F. Deissenboeck, M. Feilkas, B. Hummel, B. Schaetz, S. Wagner, C. Domann, and J. Streit. Can clone detection support quality assessments of requirements specifications? In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 2*, pages 79–88, 2010.

[58] M. Juuti, S. Szyller, S. Marchal, and N. Asokan. Prada: protecting against dnn model stealing attacks. In *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 512–527. IEEE, 2019.

[59] 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.

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

[61] 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.

[62] 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, July 2002.

[63] 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.

[64] 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.

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

[66] R. Komondoor and S. Horwitz. Using slicing to identify duplication in source code. In *Proceedings of International Static Analysis Symposium*, pages 40–56. Springer, 2001.

[67] 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.

[68] S. Kornblith, M. Norouzi, H. Lee, and G. Hinton. Similarity of neural network representations revisited. In *International Conference on Machine Learning*, pages 3519–3529. PMLR, 2019.

[69] R. Koschke, R. Falke, and P. Frenzel. Clone detection using abstract syntax suffix trees. In *Proceedings of 13th Working Conference on Reverse Engineering, 2006 (WCRE'06)*, pages 253–262. IEEE, 2006.

[70] U. Kose, O. Deperlioglu, J. Alzubi, and B. Patrut. *Deep learning for medical decision support systems*. Springer, 2021.

[71] D. Koznov, D. Luciv, H. A. Basit, O. E. Lieh, and M. Smirnov. Clone detection in reuse of software technical documentation. In *International Andrei Ershov Memorial Conference on Perspectives of System Informatics*, pages 170–185. Springer, 2015.

[72] J. Krinke. Identifying similar code with program dependence graphs. In *Proceedings of the Eighth Working Conference on Reverse Engineering*, pages 301–309. IEEE, 2001.

[73] D. E. Krutz and W. Le. A code clone oracle. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, MSR 2014, pages 388–391, New York, NY, USA, 2014. ACM.

[74] D. E. Krutz and E. Shihab. Cccd: Concolic code clone detection. In *2013 20th Working Conference on Reverse Engineering (WCRE)*, pages 489–490, Oct 2013.

[75] S. Kullback and R. A. Leibler. On information and sufficiency. *The annals of mathematical statistics*, 22(1):79–86, 1951.

[76] A. Laakso and G. Cottrell. Content and cluster analysis: assessing representational similarity in neural systems. *Philosophical psychology*, 13(1):47–76, 2000.

[77] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[78] H. Li and S. Thompson. Similar code detection and elimination for erlang programs. In *International Symposium on Practical Aspects of Declarative Languages*, pages 104–118. Springer, 2010.

[79] 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.

[80] Y. Li, J. Yosinski, J. Clune, H. Lipson, J. E. Hopcroft, et al. Convergent learning: Do different neural networks learn the same representations? In *FE@ NIPS*, pages 196–212, 2015.

[81] C. Liu, C. Chen, J. Han, and P. S. Yu. Gplag: detection of software plagiarism by program dependence graph analysis. In *Proceedings of the 12th ACM SIGKDD International conference on Knowledge Discovery and Data mining*, pages 872–881. ACM, 2006.

[82] A. Maven. http://maven.apache.org/.

[83] 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.

[84] G. Menardi and N. Torelli. Training and assessing classification rules with imbalanced data. *Data mining and knowledge discovery*, 28(1):92–122, 2014.

[85] G. Montavon and K.-R. Müller. Better representations: Invariant, disentangled and reusable. In *Neural Networks: Tricks of the Trade*, pages 559–560. Springer, 2012.

[86] A. S. Morcos, M. Raghu, and S. Bengio. Insights on representational similarity in neural networks with canonical correlation. *arXiv preprint arXiv:1806.05759*, 2018.

[87] L. A. Neubauer. Kamino: Dynamic approach to semantic code clone detection. *Technical Report, Department of Computer Science, Columiba University, CUCS-022-14*, 2015.

[88] A. Odena, C. Olsson, D. Andersen, and I. Goodfellow. Tensorfuzz: Debugging neural networks with coverage-guided fuzzing. In *International Conference on Machine Learning*, pages 4901–4911. PMLR, 2019.

[89] S. J. Oh, B. Schiele, and M. Fritz. Towards reverse-engineering black-box neural networks. In *Explainable AI: Interpreting, Explaining and Visualizing Deep Learning*, pages 121–144. Springer, 2019.

[90] R. Pan and H. Rajan. On decomposing a deep neural network into modules. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 889–900, 2020.

[91] 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.

[92] K. Pei, Y. Cao, J. Yang, and S. Jana. Deepxplore: Automated whitebox testing of deep learning systems. In *proceedings of the 26th Symposium on Operating Systems Principles*, pages 1–18, 2017.

[93] H. V. Pham, S. Qian, J. Wang, T. Lutellier, J. Rosenthal, L. Tan, Y. Yu, and N. Nagappan. Problems and opportunities in training deep learning software systems: an analysis of variance. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, pages 771–783, 2020.

[94] M. Raghu, J. Gilmer, J. Yosinski, and J. S. Dickstein. Svcca: Singular vector canonical correlation analysis for deep understanding and improvement. *network*, 200(200):200, 2017.

[95] C. Ragkhitwetsagul, J. Krinke, and D. Clark. A comparison of code similarity analysers. *Empirical Software Engineering*, 23(4):2464–2519, Aug 2018.

[96] F. Rahman, C. Bird, and P. Devanbu. Clones: what is that smell? *Empirical Software Engineering*, 17(4):503–530, Aug 2012.

[97] S. Rastogi, K. Bhushan, and B. Gupta. Android applications repackaging detection techniques for smartphone devices. *Procedia Computer Science*, 78:26–32, 2016. 1st International Conference on Information Security Privacy 2015.

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

[99] P. Refaeilzadeh, L. Tang, and H. Liu. Cross-validation. In *Encyclopedia of database systems*, pages 532–538. Springer, 2009.

[100] R. N. Reith, T. Schneider, and O. Tkachenko. Efficiently stealing your machine learning models. In *Proceedings of the 18th ACM workshop on privacy in the electronic society*, pages 198–210, 2019.

[101] A. Roy, J. Sun, R. Mahoney, L. Alonzi, S. Adams, and P. Beling. Deep learning detecting fraud in credit card transactions. In *2018 Systems and Information Engineering Design Symposium (SIEDS)*, pages 129–134. IEEE, 2018.

[102] C. K. Roy and J. R. Cordy. A survey on software clone detection research. *Technical Report, Queen's University at Kingston*, 2007.

[103] C. K. Roy and J. R. Cordy. A survey on software clone detection research. *Queen's School of Computing TR*, 541(115):64–68, 2007.

[104] C. K. Roy and J. R. Cordy. Nicad: Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization. In *Proceedings of the 16th IEEE International Conference on Program Comprehension (ICPC08)*, pages 172–181. IEEE, 2008.

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

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

[107] V. Saini, F. Farmahinifarahani, Y. Lu, P. Baldi, and C. V. Lopes. Oreo: Detection of clones in the twilight zone. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 354–365, 2018.

[108] V. Saini, F. Farmahinifarahani, Y. Lu, P. Baldi, and C. V. Lopes. Oreo: Detection of clones in the twilight zone. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2018, pages 354–365, New York, NY, USA, 2018. ACM.

[109] 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.

[110] H. Sajnani. *Large-Scale Code Clone Detection*. PhD thesis, University of California, Irvine, 2016.

[111] 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 (ICSE16)*, pages 1157–1168. IEEE, 2016.

[112] I. Samoladas, G. Gousios, D. Spinellis, and I. Stamelos. The sqo-oss quality model: measurement based open source software evaluation. In *Proceedings of the International Conference on Open Source Systems*, pages 237–248. 2008.

[113] S. Schulze and M. Kuhlemann. Advanced analysis for code clone removal. In *Proceedings des Workshops der GI-Fachgruppe Software Reengineering (SRE), erschienen in den GI Softwaretechnik-Trends 29 (2)*, pages 10–12. Citeseer, 2009.

[114] 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.

[115] A. Sheneamer and J. Kalita. A survey of software clone detection techniques. *International Journal of Computer Applications*, 137(10):1–21, 2016.

[116] A. Sherry and R. K. Henson. Conducting and interpreting canonical correlation analysis in personality research: A user-friendly primer. *Journal of personality assessment*, 84(1):37–48, 2005.

[117] P. Shrout and J. Fleiss. Intraclass correlations: Uses in assessing rater reliability. *Psychological Bulletin*, 86(2):420–428, 1979.

[118] Songhao Wu. 3 Best metrics to evaluate Regression Model? `https://towardsdatascience.com/what-are-the-best-metrics-to-evaluate-your-regression-model-418ca481755b`, May 2020.

[119] M. Srikar Yellapragada. Are the proposed similarity metrics also a measure of functional similarity? Master's thesis, New York University, 2020.

[120] 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.

[121] 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.

[122] 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.

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

[124] 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.

[125] 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.

[126] J. Svajlenko and C. K. Roy. Fast and flexible large-scale clone detection with cloneworks. In *Proceedings of the 39th International Conference on Software Engineering Companion*, pages 27–30. IEEE Press, 2017.

[127] V. Svetnik, A. Liaw, C. Tong, J. C. Culberson, R. P. Sheridan, and B. P. Feuston. Random forest: a classification and regression tool for compound classification and qsar modeling. *Journal of chemical information and computer sciences*, 43(6):1947–1958, 2003.

[128] Y. Tang, R. Khatchadourian, M. Bagherzadeh, R. Singh, A. Stewart, and A. Raja. An empirical study of refactorings and technical debt in machine learning systems. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 238–250. IEEE, 2021.

[129] 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.

[130] F. Tramèr, F. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart. Stealing machine learning models via prediction apis. In *25th {USENIX} Security Symposium ({USENIX} Security 16)*, pages 601–618, 2016.

[131] 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.

[132] H. Wang, Z. Wang, Y. Guo, and X. CHEN. Detecting repackaged android applications based on code clone detection technique. *SCIENTIA SINICA Informationis*, 44(1):142–157, 2014.

[133] L. Wang, L. Hu, J. Gu, Y. Wu, Z. Hu, K. He, and J. Hopcroft. Towards understanding learning representations: To what extent do different neural networks learn the same representation. *arXiv preprint arXiv:1810.11750*, 2018.

[134] 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.

[135] S. Wang, T. Liu, and L. Tan. Automatically learning semantic features for defect prediction. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, pages 297–308. IEEE, 2016.

[136] 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.

[137] X. Wang, Y.-C. Jhi, S. Zhu, and P. Liu. Detecting software theft via system call based birthmarks. In *2009 Annual Computer Security Applications Conference*, pages 149–158. IEEE, 2009.

[138] 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.

[139] P. Weissgerber and S. Diehl. Identifying refactorings from source-code changes. In *21st IEEE/ACM International Conference on Automated Software Engineering (ASE'06)*, pages 231–240, Sept 2006.

[140] 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*, pages 87–98. ACM, 2016.

[141] M. White, C. Vendome, M. Linares-Vásquez, and D. Poshyvanyk. Toward deep learning software repositories. In *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*, pages 334–345. IEEE, 2015.

[142] 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.

[143] H. Xiao, K. Rasul, and R. Vollgraf. Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms. *arXiv preprint arXiv:1708.07747*, 2017.

[144] X. Xie, L. Ma, F. Juefei-Xu, M. Xue, H. Chen, Y. Liu, J. Zhao, B. Li, J. Yin, and S. See. Deephunter: a coverage-guided fuzz testing framework for deep neural networks. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 146–157, 2019.

[145] X. Xie, L. Ma, H. Wang, Y. Li, Y. Liu, and X. Li. Diffchaser: Detecting disagreements for deep neural networks. In *IJCAI*, pages 5772–5778, 2019.

[146] M. Xu, J. Liu, Y. Liu, F. X. Lin, Y. Liu, and X. Liu. A first look at deep learning apps on smartphones. In *The World Wide Web Conference*, pages 2125–2136, 2019.

[147] T. Yamashina, H. Uwano, K. Fushida, Y. Kamei, M. Nagura, S. Kawaguchi, and H. Iida. Shinobi: A real-time code clone detection tool for software maintenance. *Nara Institute of Science and Technology*, page 26, 2008.

[148] W. Yang. Identifying syntactic differences between two programs. *Software: Practice and Experience*, 21(7):739–755, 1991.

[149] M. F. Zibran and C. K. Roy. Towards flexible code clone detection, management, and refactoring in ide. In *Proceedings of the 5th International Workshop on Software Clones*, pages 75–76, 2011.