

Towards Automating Precision Studies of Clone Detectors

Vaibhav Saini[†] Farima Farmahinifarahani[†] Yadong Lu[†]
Di Yang[†] Pedro Martins[†] Hitesh Sajjani[‡]
Pierre Baldi[†] Cristina V. Lopes[†]
[†] UC Irvine, USA [‡] Microsoft, USA

{vpsaini, farimaf, yadongl1, diy4, pribeiro, pfbaldi, lopes}@uci.edu
hitsaj@microsoft.com

Abstract—Current research in clone detection suffers from poor ecosystems for evaluating precision of clone detection tools. Corpora of labeled clones are scarce and incomplete, making evaluation labor intensive and idiosyncratic, and limiting inter-tool comparison. Precision-assessment tools are simply lacking.

We present a semi-automated approach to facilitate precision studies of clone detection tools. The approach merges automatic mechanisms of clone classification with manual validation of clone pairs. We demonstrate that the proposed automatic approach has a very high precision and it significantly reduces the number of clone pairs that need human validation during precision experiments. Moreover, we aggregate the individual effort of multiple teams into a single evolving dataset of labeled clone pairs, creating an important asset for software clone research.

Index Terms—Precision Evaluation, Clone Detection, Machine learning, Open source labeled datasets

I. INTRODUCTION

Source code clone detection is the task of finding similar software pieces, according to a certain concept of similarity. These pieces can be statements, blocks of code, functions, classes, or even complete source files, and their similarity can be syntactic, semantic or both. Cloning in software source code is as ubiquitous as software itself, which gives clone detection tools many applications: plagiarism and copyrights enforcement [1], [2], detection of errors/faults/bugs [2], code optimization and refactoring [3], [4], analysis of programmers behaviors [5] or program understanding [2] are some examples.

In a systematic literature review, Rattan et al. found at least 70 clone detection tools and techniques [6]. Clone detectors differ substantially in the underlying techniques and scope of application. In terms of technical approach used, one can find techniques that are learning-based [7], [8], token-based [9]–[11], tree-based [12], [13], graph-based [14], or text-based [15]. With respect to the scope, one can find tools that are language-specific and language-agnostic, with varying degrees of specialization. While there are many tools and techniques published to detect clones, not much effort is spent on streamlining the evaluation of these tools and techniques.

The effectiveness of clone detection tools is usually evaluated in terms of precision and recall. Precision is the percentage of true positives (clone pairs) within a set of code pieces identified by the tool as clones. Recall is the percentage of true positives

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

In the field of code clone detection, building labeled datasets is particularly challenging and requires software expertise. Methods can exist inside methods, or they can vary wildly in size, scope, semantics and nature. In addition, to manually validate all the possible clones would require quadratic comparisons, a combinatorial problem that becomes infeasible with growing codebases.

For this reason, most datasets used in code clone studies are either small or synthetically created or are labeled only for a subset of pairs. The dataset by Bellon *et al.* [16], the dataset by Murakami *et al.* [17], SOCO 2014 [18] or BigCloneBench [19] have one of the above mentioned limitations.

There has been good progress in measuring recall systematically of clone detection tools. Based on BigCloneBench dataset [19], BigCloneEval [20] estimates recall automatically by measuring how many of the labeled clone pairs are included in the output of a clone detector. However, BigCloneEval stops short of estimating precision because BigCloneBench does not contain labels for *all* possible clone pairs in it. If a clone detector identifies a clone pair that is not marked as such, only manual inspection can tell whether the pair is a false positive or a true positive. Since manual inspection is a difficult, labor intensive, and time consuming task, most clone detection approaches estimate their precision by sampling a number of their reported clone pairs, and then manually inspecting the sampled set [8], [10], [21]. So, while clone detectors report recall using BigCloneEval, the determination of their precision is still a subjective and a manual process, leading to difficulties

in comparing with other tools. The lack of an established labeled dataset for precision creates a number of problems [10]: (i) precision estimation can suffer from sampling bias when sample set is small and not representative of the population, (ii) large manual effort is required to conduct precision studies. As each pair in the sample needs to be evaluated, often by multiple judges, the total manual effort required to complete a precision study is substantial, and (iii) the efforts put into the manual inspection of clone pairs are not reused. Each time authors want to estimate their tool’s precision, they typically start from scratch.

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

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

The main contributions of this work are the following: (i) a semi-automated, high-precision, approach for classification of clones that reduces the manual effort of precision studies significantly, (ii) a publicly available web application that enables clone detection researchers to conduct precision experiments of clone detection tools and techniques, and (iii) an evolving dataset of manually validated clone pairs that is publicly available. With time, as InspectorClone is used, the number of humanly validated clone pairs will increase in the dataset.

This work is organized as follows. In Section II, we present InspectorClone, the tool implementation of our approach. The automatic mechanisms of clone pair resolution, and related concepts are elaborated in Section III, and then it is evaluated in Section IV. Section V presents related work, and threats to validity are explained in Section VI. Finally, we present the conclusions and future work in Section VII.

II. INSPECTORCLONE

Measuring the precision of a clone detector is not a trivial process. To measure the precision, one can choose to manually validate all the clone pairs reported by a clone detector. This process, however, is extremely time consuming and impractical as the number of clone pairs reported by a tool on a standard dataset like BigCloneBench is in millions. A more practical process is to estimate the precision by humanly validating a random and statistically significant sample of clone pairs. This is what researchers do to estimate the precision of clone detectors [8], [10], [21], [22]. In this process, after running a

clone detector on a dataset and getting the clone pairs, a random and statistically significant sample set of these clone pairs is assigned to multiple judges for manual inspection. The judges examine each pair to decide if it is a true clone and/or what type of clone it is. When all sampled pairs have been validated by all judges, researchers aggregate the judges’ decisions, usually by taking the majority vote, and report precision.

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

InspectorClone conducts precision studies on the dataset curated by Svajlenko et al. for facilitating recall studies [20]. Svajlenko et al. curated this dataset using IJaDataset-2.0 to conduct recall study using BigCloneEval. The dataset is available for download on InspectorClone’s website. InspectorClone does not run the clone detection tool; instead, it expects users to upload the clone pairs reported by their tool to the website. The work flow is as follows.

A user, John, registers himself and his tool into InspectorClone. After registration, John can download the dataset of source code and run it on his clone detector. John, then uploads the clone pairs to InspectorClone where InspectorClone filters out the methods that are less than 50 tokens, a standard filter used in precision studies [10], [22]. John now creates an experiment to estimate the precision of his tool. He then invites multiple judges to evaluate the pairs. Once the judges are invited, InspectorClone selects a random and statistically significant sample of clone pairs. From this sample InspectorClone tries to automatically validate as many pairs as it can. All of the remaining pairs of the sample, which InspectorClone did not resolve, are then shown to the judges.

When a judge, Alice, starts an experiment assigned to her, she is shown a web page as shown in Figure 1. All unresolved pairs will be shown to her. This page is composed by a split screen with two columns, showing both members of a pair. The code is syntax highlighted to increase the readability. Alice must then decide if this pair does indeed represent a clone or not (if it is a true or a false positive). There are two optional form elements: one to select the clone type, and another to leave a comment.

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

(143 more to evaluate)

<pre> 1 public boolean validateVersion_0_1() throws Exception { 2 String testRoot = TEST_Version_0_1; 3 boolean test = true; 4 sReferencePS.clear(); 5 sReferencePS.set("body", "bodyCheck"); 6 sReferencePS.set("init", "initCheck"); 7 sReferencePS.set("support", "supportCheck"); 8 sTestPS.load(new File(getTestFolder(), PREFIX_RESULT + testRoot)); 9 test = sTestPS.contains(sReferencePS) && test; 10 if (!test) { 11 displayPropertySets(sTestPS, sReferencePS); 12 } 13 return test; 14 } </pre>	<pre> 1 public boolean validateDirectiveSection() throws Exception { 2 String testRoot = TEST_DirectiveSection; 3 boolean test = true; 4 sReferencePS.clear(); 5 sReferencePS.set("list0", EMPTY); 6 sReferencePS.set("list1", "[0]"); 7 sReferencePS.set("list2", "[0][1]"); 8 sReferencePS.set(FOO, BAR); 9 sTestPS.load(new File(getTestFolder(), PREFIX_RESULT + testRoot)); 10 test = sTestPS.contains(sReferencePS) && test; 11 if (!test) { 12 displayPropertySets(sTestPS, sReferencePS); 13 } 14 return test; 15 } </pre>
<p>Vote:</p> <p><input type="radio"/> True Positive <input type="radio"/> False Positive</p>	<p>Clone type:</p> <p><input type="checkbox"/> Type 2 <input type="checkbox"/> Type 3 <input type="checkbox"/> Type 4</p>
<p>Explanation:</p> <div style="border: 1px solid #ccc; height: 20px; width: 100%;"></div>	
<p>Submit</p>	

Fig. 1: Validation of clone candidates on InspectorClone.

of humanly judged pairs to increase in this database, thereby creating a valuable asset for the community.

III. AUTOMATIC CLASSIFICATION OF CLONES

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

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

A. Definitions

In this section, we elaborate on the terms and definitions that are pivotal to discussing InspectorClone's mechanisms.

Clone Pair: A pair of code fragments that are similar, specified by the triple (f_1, f_2, ϕ) , including the similar code fragments f_1 and f_2 , and their clone type ϕ [22].

Clone Types: Based on the literature [2], our work uses the following four types of source code clones, the first three being similar on the textual and syntactic level, and the fourth type defining similarity on the functional, semantic level:

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

Type II: Identical code fragments, except for differences in identifier names and literal values, as well as Type I differences.

Type III: Syntactically similar code fragments that differ at the statement level. The fragments have statements added, modified and/or removed with respect to each other, in addition to Type I and Type II clone differences

Type IV: Syntactically dissimilar code fragments that implement the same functionality.

The definition to classify clones as Type III does not specify what should be the minimum syntactical similarity between the methods of a clone pair to be classified as Type III. Also, the lack of consensus in the community of clone researchers about this similarity makes it difficult to separate Type IV and Type III clones. To address this issue, the popular clone benchmark, BigCloneBench [19], [22], has divided the zone between Type III and Type IV into four subcategories based on syntactical similarity values: Very Strongly Type III (VST3) with similarity in range of [0.9, 1.0), Strongly Type III (ST3) with similarity being in [0.7,0.9), Moderately Type III (MT3) with similarity in range of [0.5, 0.7), and Weakly Type III (WT3/4) having similarity in the range of [0.0,0.5). More details about these subcategories can be found elsewhere [19].

Listing 1: Example: Action Tokens

```

1 public Enumeration children () {
2   Enumeration allChildren = super.children ();
3   Vector filtered = new Vector ();
4   DiligentNode node;
5   while ( allChildren.hasMoreElements () ) {
6     node = (DiligentNode) allChildren.nextElement ();
7     if (!node.isFiltered (true)) filtered.addElement (node);
8   }
9   return filtered.elements ();
10 }

```

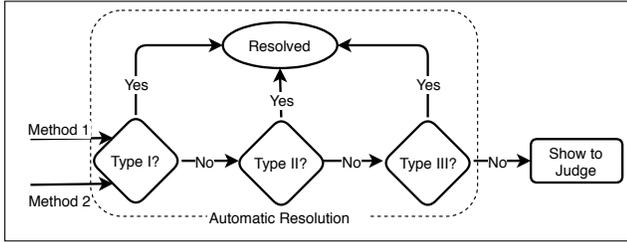


Fig. 2: The pipeline for clone validation.

Action Token: Action tokens of a method are the tokens corresponding to the methods called and class fields accessed by that method [8]. Additionally, the array accesses made by a method are also special Action tokens namely *ArrayAccess* and *ArrayAccessBinary*, where array access of kind $arr[i]$ is an *ArrayAccess* Action token and $arr[i+1]$ is an *ArrayAccessBinary* Action token. In the code provided in Listing 1, Action tokens are: *children()*, *hasMoreElements()*, *nextElement()*, *isFiltered()*, *addElement()*, and *elements()*.

Action Filter: A filter which ensures a minimum amount of similarity between the Action tokens of two methods [8].

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 $\langle t, freq \rangle$, where t is the Action Token and $freq$ is the number of appearances of this token in the method. M_1 and M_2 satisfy the Action filter if $\frac{Sim(A_1, A_2)}{\max(|A_1|, |A_2|)} \geq \theta$, where θ is Action filter threshold such that $0 \leq \theta \leq 1$.

B. Overview of the Approach

The methodology for clone resolution follows the pipeline presented in Figure 2. The two methods in a candidate pair¹ go through a series of steps in which they are checked against a certain clone type. If at any step a pair is evaluated as a true clone pair, it is marked as a true positive and the system proceeds to the next candidate pair. Otherwise, if all of the steps are failed to evaluate a pair as a true positive, the pair is presented to the human judge for manual inspection. These steps are ordered by computational complexity for system performance (and also by increasing clone type complexity), and are individually described in the next sections.

¹A candidate pair consists of two piece of code reported as clone pair by a clone detection tool. Our approach validates these pairs, and only when they are resolved as true positives they are called clone pairs.

C. Automatic Resolution of Type I Clones

As described in Section III-A, two pairs are Type I clones if they are exact replicas when neglecting source code comments and layout². This makes the validation of Type I candidates similar to a simple string comparison after removing certain elements. We use Algorithm 1 to check if a candidate pair is a Type I clone. Starting with a candidate pair, the algorithm, first, removes all source code comments from both method bodies (lines 2 and 3), then removes white spaces and newlines from them (lines 4 and 5), and finally computes and compares the Hash (SHA-256), of both method bodies (line 6).

Algorithm 1 Automatic Type I Resolution

INPUT: M_1 and M_2 are strings representing the method bodies (including method signature) of two methods for which we want to know if they are Type I clones.
OUTPUT: *Boolean*

```

1: function ISATYPEONE(M1, M2)
2:   M1 = REMOVECOMMENTS(M1)
3:   M2 = REMOVECOMMENTS(M2)
4:   M1 = REMOVEWHITESPACESANDNEWLINES(M1)
5:   M2 = REMOVEWHITESPACESANDNEWLINES(M2)
6:   return HASH(M1)==HASH(M2)
7: end function

```

D. Automatic Resolution of Type II Clones

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

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

Metric heuristic: Software metrics, measuring different characteristics of source code, can capture structural information of a method. These measurements are resilient to changes in identifier names and literals – a useful property in the detection of Type II clones. Hence, we use 24 method level software metrics shown in Table I for Type II resolution. The details of these metrics can be found elsewhere [8], [23]. A detailed explanation about the application of Action tokens and software metrics in clone detection can be found in [8].

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

This algorithm ensures that a candidate pair is resolved as Type II only when there is a 100% match in both the metrics and the Action tokens. The rational is that Type II clones

²The syntax of Java is not dependent on layout, so we can ease the definition of Type I clones. For layout-dependent syntaxes like the ones found in Python or Haskell, this approach would require a more careful deliberation

TABLE I: Method-Level Software Metrics from [8]

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	CREP	# 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

differ in identifier names and literal values while their structure (captured by metrics), and their method calls and accessed class fields (captured using Action tokens) remain the same.

Algorithm 2 Automatic Type II Resolution

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

```

1: function ISYPTWO( $M1, M2$ )
2:    $ListATofM1 = GETACTIONTOKENS(M1)$ 
3:    $ListATofM2 = GETACTIONTOKENS(M2)$ 
4:   if IDENTICAL( $ListATofM1, ListATofM2$ ) then
5:      $ListMetM1 = GETMETRICS(M1)$ 
6:      $ListMetM2 = GETMETRICS(M2)$ 
7:     return IDENTICAL( $ListMetM1, ListMetM2$ )
8:   end if
9:   return False
10: end function

```

E. Automatic Resolution of Type III Clones

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

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

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

1) *Dataset Curation:* Since the machine learning classifier is supposed to resolve Type III candidates, we need a training

TABLE II: Dataset Creation Process Statistics

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

set with clone pairs from this category. Also, to resolve clone pairs with very high precision, we want our dataset to contain true clone pairs which are very similar in terms of both their semantics and structure. To generate such dataset, we use two token based state of the art clone detectors, CloneWorks (Aggressive mode) [11] and SourcererCC [10]. The configurations of these tools are shown in Table VI. These clone detectors detect Type III clone pairs up to ST3 category, where the methods in each clone pair have high structural similarity. On the other hand, we ensure high semantic similarity in the methods of each clone pair in our dataset by using Action Filter with threshold set to 90%. The starting dataset used to generate our training dataset is BigCloneBench.

The numbers related to dataset creation process are reported in Table II. The training set includes equal number of both assumed positives (clones) and assumed negatives (non-clones). To get the set of assumed positives, we took an intersection of the clone pairs detected by the two tools (RowId 3 in Table II). We then removed all Type I and Type II pairs from this intersection and selected the pairs which satisfy our Action filter (RowId 4). To ensure that these pairs are true clone pairs, we randomly sampled 1,851 pairs (a statistically significant sample with 99% confidence level and 3% confidence interval), and validated them manually. Two judges, who are also the authors of this paper, independently went through these clone pairs and unanimously found all pairs to be true clone pairs.

Listing 2 shows an example of true clone pair (VST3) found by the judges. Both methods in this example seem to have a very similar aim: first, they fill a *LinkedList* (line 6 and 26) and then they iterate over the *LinkedList* to remove its contents (lines 14 to 16 and lines 34 to 36). The methods not only share many Action tokens, their structures also look very similar. Moreover, the line and token similarity between the methods are high, making this pair a good example of a true positive.

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

To get such assumed negative pairs, we modified OreO [8], a clone detector designed to detect Type III clones even in

Listing 2: Example VST3 Clone Pair

```

1 private boolean doPurgeStudy(DirWriter w, DirRecord parent, int []
2 counter) throws IOException {
3     boolean matchAll = true;
4     LinkedList toRemove = new LinkedList();
5     for (DirRecord rec = parent.getFirstChild(true); rec != null; rec
6         = rec.getNextSibling(true)) {
7         if (doPurgeSeries(w, rec, counter)) {
8             toRemove.add(rec);
9         } else {
10            matchAll = false;
11        }
12    }
13    if (matchAll) {
14        return true;
15    }
16    for (Iterator it = toRemove.iterator(); it.hasNext(); ) {
17        counter[0] += w.remove((DirRecord) it.next());
18    }
19    return false;
20 }
21 -----
22 private boolean doPurgeInstances(DirWriter w, DirRecord parent, int []
23 counter) throws IOException {
24     boolean matchAll = true;
25     LinkedList toRemove = new LinkedList();
26     for (DirRecord rec = parent.getFirstChild(true); rec != null; rec
27         = rec.getNextSibling(true)) {
28         File file = w.getRefFile(rec.getRefFileIDs());
29         if (!file.exists()) {
30             toRemove.add(rec);
31         } else {
32             matchAll = false;
33         }
34    }
35    if (matchAll) {
36        return true;
37    }
38    for (Iterator it = toRemove.iterator(); it.hasNext(); ) {
39        counter[0] += w.remove((DirRecord) it.next());
40    }
41    return false;
42 }

```

harder clone categories, to predict non-clones such that they have at least 90% similarity in their Action tokens (RowId 5). The original source code of Oreo is available at [24]. Then, we took a union of the clone pairs reported by three state of the art clone detectors: CloneWorks, SourcererCC, and NiCad [15] (RowId 6). NiCad’s configurations are shown in Table VI. To ensure high confidence in the non-clone pairs, we removed any non-clone pair which is present in the union set (RowId 7). Finally, we did a manual analysis similar to what we did for true positives to gain more assurance about the non-clone pairs. The same two judges, independently as before, went through a random sample of 400 pairs. They found many examples which were definitely non-clones, and also found some examples of MT3 and WT3/4 clones, where the pairs shared high semantic similarity but the structural similarity was weak. This is useful in increasing the precision of the machine learning model since it learns to classify these harder pairs, which are closer to the threshold boundary, as non clones. This is a desirable behavior as these harder cases are then left for human judgment.

Listing 3 shows an example of an MT3 pair found by the judges. Both methods in this pair are semantically similar as they both intend to copy the contents from an *InputStream* to an *OutputStream*. The structural and token similarity between the two methods, however, is low, making it harder to detect

Listing 3: Example MT3 Clone Pair

```

1 public static void copy(InputStream i, int buf, OutputStream o)
2     throws IOException {
3     byte b[] = new byte[buf];
4     for (; ; ) {
5         int g = i.read(b);
6         if (g == -1) break;
7         o.write(b, 0, g);
8     }
9 }
10 -----
11 public static void copyTo(InputStream in, OutputStream out) throws
12     IOException {
13     byte buffer[] = new byte[2048];
14     int n = 0;
15     while ((n = in.read(buffer)) != -1) {
16         out.write(buffer, 0, n);
17     }
18     out.flush();
19 }

```

Listing 4: Example WT3/4 Clone Pair

```

1 protected void copy(InputStream _in, OutputStream _out) throws
2     IOException {
3     byte[] buf = new byte[1024];
4     int len = 0;
5     while ((len = _in.read(buf)) > 0) _out.write(buf, 0, len);
6 }
7 -----
8 public static long copy(InputStream is, OutputStream os) throws
9     IOException {
10    byte buffer[] = new byte[1024];
11    int readed;
12    long length = 0;
13    while ((readed = is.read(buffer)) != -1) {
14        if (readed > 0) {
15            length += readed;
16            os.write(buffer, 0, readed);
17        } else {
18            TimerHelper.notSafeSleep(100);
19        }
20    }
21    return length;
22 }

```

as a clone pair by many token based clone detectors. Similarly, Listing 4 shows another pair that semantically, are performing the same task, but the structural similarity between the two methods is very low. Such methods are good candidates that should be left for human judgment.

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

2) *Deep Learning Model*: To classify Type III clones we are using Siamese architecture to train a Deep Neural Networks (DNN) model. Siamese models are well suited for problems where two things need to be compared against each other, for example comparing fingerprints [25]. Also, in a recent work on clone detection, Saini et al. compared different architectures of Deep Neural Networks (DNN) and found Siamese DNN to outperform the other DNN architectures [8]. We also carried

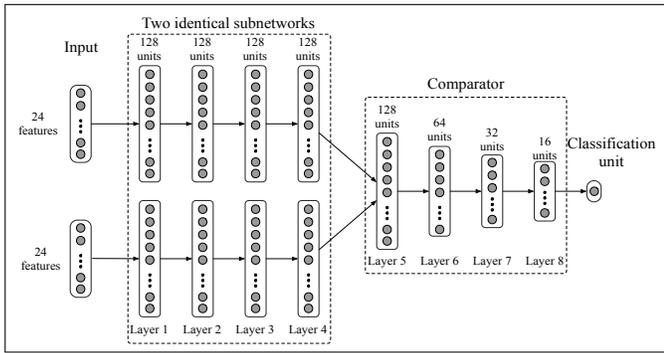


Fig. 3: Siamese architecture

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

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

out model comparison analyses on the train dataset at hand and found Siamese model to outperform other models. Model comparison results are explained later in this section.

Figure 3 shows the architecture of the Siamese model we trained for our approach. It consists of three components: i) two identical subnetworks, ii) a comparator unit, and iii) a classification unit. The input to the model are the feature vectors of each method (24 metrics) in the candidate pair. These feature vectors are then transformed and processed by the two subnetworks and the comparator. Finally, the classification unit outputs a number between 0 and 1, representing the probability of a pair being a clone. A more detailed explanation of this architecture and its components can be found elsewhere [8].

To perform the model selection experiments, the train dataset of 116,000 code pairs is randomly divided into 80% for training and 20% for testing. Furthermore, 5,000 pairs from the training set are set aside for validation purposes, i.e. for hyperparameter tuning. First, we explored the hyper-parameter tuning for the Siamese model. The best performing model is the one shown in Figure 3. Each of the two subnetwork layers in this model has 4 layers, each with 128 neurons and the comparator has four fully connected layers of sizes 128-64-32-16. The output of the comparator is then fed into a single classification neuron with sigmoidal (logistic) activation function. The output of this neuron is a value between 0 and 1. Normally, the values more than 0.5 are assigned label 1, and values less than 0.5, are assigned label 0. However, since our goal in this problem is to achieve almost perfect precision (so that the number of false positives tends to zero), during production (testing), we set the threshold to announce a pair a true clone pair to be 0.9. The code pairs which have prediction values between 0 to 0.9 are sent to human judges for further inspection. Table III shows the Precision and Recall values on the test set at different thresholds. As it is observed, using this deep learning approach

TABLE IV: Precision and Recall values on test set

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

with a threshold of 0.9 yields almost perfect precision (0.996).

We also compared the Siamese DNN to other models with different architectures, including: (1) a plain fully connected neural network (Plain DNN) with similar number and sizes of hidden layers as the Siamese one; (2) a shallow neural network (Shallow NN) with one hidden layer and similar number of parameters; and (3) a logistic regression model. Since our final goal is to achieve a high Precision, all the comparisons are done when the thresholds for all models are set to be 0.9. We first compared their performance during the training stage. Figure 4 shows that the Siamese DNN outperforms the other models in terms of accuracy on the validation set. The accuracy of the Siamese DNN converges to 96.0%, while the accuracy for plain DNN and Shallow NN converges only to 94.0%. Figure 5 shows that for the validation loss also, the Siamese structure is superior to the other models. The average loss value for the Siamese DNN converges to 0.103 (as apposed to Plain DNN: 0.139; Shallow NN: 0.162; Logistic regression model: 0.222). Thus in short, the Siamese DNN better fits the training data.

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

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

Methodology. We used the clone pairs reported by SourcererCC on the BigCloneBench Dataset. We ran InspectorClone with four different threshold values of Action Filter: 60%, 65%, 70%, and 75%. One author manually inspected the clone pairs that are automatically resolved by InspectorClone to figure out the number of false positives in them. Results of this analysis are denoted in Table V. The first column of this table shows the examined thresholds, and the next three columns, respectively, denote the number of automatically resolved Type I, Type II, and Type III clone pairs. The next three columns show the number of false positives observed at each clone category, and the last column depicts the number of clone pairs that need the manual validation by humans. At 60% and 65% thresholds, we observed some false positives, whereas at 70% and 75% thresholds, no false positives were observed. The number of automatically resolved clone pairs at 70% threshold (49) is greater than this number at 75% threshold (18). Consequently, 70% threshold was selected to be used in Action Filter.

IV. EVALUATION

As discussed earlier, the main goal of our approach is to automatically resolve as much as clone pairs as possible with

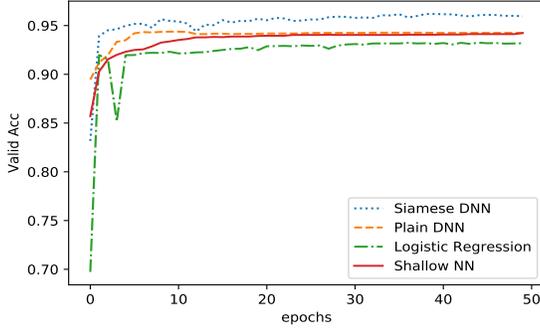


Fig. 4: Validation Accuracy

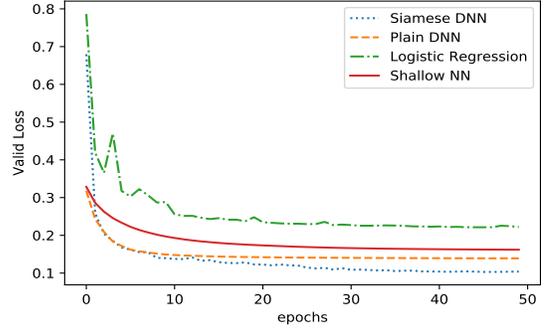


Fig. 5: Validation Loss

TABLE V: Sensitivity Analysis Statistics

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

TABLE VI: Reduction of Manual Effort

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

high precision. Hence, to evaluate it, we designed an experiment using InspectorClone, and seven clone detectors. The goal of this experiment is twofold: i) to understand the impact of our approach on the reduction of manual effort, and ii) to measure the precision of the automatic clone resolution approach.

We include SourcererCC, iClones [26], NiCad, CloneWorks, SimCad [27] as popular examples of modern clone detectors that support Type III clone detection. CloneWorks comes in two different modes, Aggressive and Conservative; we tested InspectorClone on both of these modes. We also include two recent tools, Oreo and CCAligner [21]. While all these tools detect clones in Type I, Type II, and early categories of Type III (*VST3* and *ST3*), Oreo and CCAligner are capable of detecting clones beyond *ST3* categories such as *MT3*. We also wanted to include Deckard [12] and CPD [28]; however, they both detect clones beyond method boundaries. At this time, we cannot reliably conduct a meaningful experiment with them on InspectorClone, which only supports method level clone detectors as of now. Also, both of these tools report their results as clone classes and not as clone pairs. When we ran processes to generate clone pairs from these clone classes, they both produced large amount of clone pairs. We killed the processes after generating more than 175G of clone pairs for each of them as these are very big files for InspectorClone to process.

We ran all tools on the recall dataset of BigCloneBench and obtained the clone pairs reported by each tool. We then uploaded the clones reported by each tool to InspectorClone, and calculated the number of clone pairs automatically resolved in each category, and the number of pairs left for manual validation. We configured InspectorClone to consider only those pairs that have methods with at least 50 language tokens, a standard size filter used in precision studies [8], [10].

To gain high confidence in our experiment results, we conducted 2 rounds of experiments for each tool (a total

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

The results of this experiment are shown in Table VI. The first column shows the name of the tool. The next three columns denote, respectively, the number of Type I, Type II, and Type III candidate pairs that were automatically resolved by InspectorClone. The fifth column shows the number of candidate pairs that could not be automatically classified, and needed manual validation by humans (out of the sample of 400). The sixth column (*FP*) contains the number of false positives (after considering majority vote) observed by human judges in the automatically resolved pairs. And finally, the

Listing 5: Example Candidate Pair from Oreo

```

1 public static <T> T readStreamAsObject(InputStream inputStream,
2   Class<T> type) throws ClassNotFoundException, IOException {
3   ObjectInputStream objectInputStream = null;
4   try {
5     objectInputStream = new ObjectInputStream(inputStream);
6     return type.cast(objectInputStream.readObject());
7   } finally {
8     Utility.close(objectInputStream);
9   }
10 }
11 -----
12 public static <T extends Serializable > T deserialise (Class<T> class1,
13   File out) throws ClassNotFoundException {
14   try {
15     FileInputStream fis = new FileInputStream(out);
16     ObjectInputStream in = new ObjectInputStream(fis);
17     Object output = in.readObject();
18     in.close();
19     return class1.cast(output);
20   } catch (IOException ex) {
21     ex.printStackTrace();
22     return null;
23   }
24 }

```

seventh column shows the configurations which were used to run the tools. These configurations are based on our discussions with their developers, and also the configurations suggested in [22]. In the table, *MIT* stands for minimum tokens, *MIL* stands for minimum number of lines, *BIN* and *IA*, respectively stand for blind identifier normalization and literal abstraction used in NiCad. Θ stands for similarity threshold (for NiCad, it is *difference threshold*, and for Oreo it is Action Filter threshold), Γ is the threshold for input partition used in Oreo. In CCaligner’s configurations, *e* stands for *edit distance*, and *q* is the *window size*. *GT* and *US* stand for *greedy transformation* and *unicode support* used in SimCad.

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

To understand why InspectorClone did not help in reducing the number of pairs for Oreo and SimCad, two judges went through the samples of one of the two experiments conducted for both tools. For SimCad, the judges reported 358 out of 400 pairs as false positives of the SimCad tool itself (10.5% precision). The presence of large number of false positives in the pairs of SimCad explains why InspectorClone did not help much in resolving its pairs. For Oreo, the judges reported a much higher precision of 80%, where they reported 80 out of 400 pairs as false positives of the tool. Almost all of the clone pairs in the sample of Oreo were found to be in harder to detect Type III categories (*MT3* and *WT3*). An example of such a pair is shown in Listing 5. Both methods in this example are reading an *object* from an input stream, and then, they cast this object into the *type* they received in their arguments. Though they are performing similar tasks and hence, are semantically similar, they differ significantly in their structural properties.

This qualifies such pairs to fall in harder to detect *MT3/WT3* categories, making them good candidates for human inspection.

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

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

In Strategy-B, 11 false positives were found, giving InspectorClone a precision score of 98.3% in Type III pairs. If pairs of all types are combined, this strategy gives a precision score of 99.57%. In their judgments of 2,545 pairs, the judges were unanimously in agreement on 2,534 pairs, giving a conservative estimate of inter-rater-reliability as 99.57%.

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

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

V. RELATED WORK

Measuring the detection capabilities of clone detection tools is an important part of source code cloning research. This demands the existence of labeled and standardized datasets that can assist with this measurement. Therefore, development

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

BigCloneBench (BCB) dataset [19], [22] is probably the most related work to ours. We have also used it in the evaluation of our approach with InspectorClone. The underlying corpus of Java source code used by BCB is IJaDataset-2.0³. This dataset represents a large inter-project Java repository containing 25,000 open source projects, with 2.3 million source files and 365M lines of code [22]. BCB contains a subset of IJaDataset curated by human judgment, and it contains over 8 million known clone pairs within IJaDataset. It is the result of using IJaDataset, selecting a series of known algorithms (sorting algorithms is one example), and tracking possible implementations of these across the dataset. Hence, not all possible clone pairs are tagged in this dataset, and there exists many pairs that are not tagged. As a result, this dataset cannot be used to measure the precision of clone detectors, but it has been used by BigCloneEval (BCE) [20] to estimate the recall of clone detector tools automatically.

Another dataset is created by Bellon *et al.* [16]. To prepare this dataset, Bellon manually validated 2% of the clones reported by then (year 2002) contemporary clone detectors for eight software systems. Svajlenko *et al.* [22] found that this benchmark is not suitable for accurate evaluation of modern clone detection tools. They attributed many of the problems in the dataset to it being built using tools that are now outdated. It has also been found to have other problems as we see next.

Murakami *et al.*'s dataset [17] is an improvement on the Bellon *et al.*'s dataset. Murakami *et al.* found out that since Bellon dataset does not contain locational information of gaped lines (i.e. lines that are present in a pair but missing in the other), it has not evaluated some Type III clones correctly. Hence, they added this information and improved the dataset with this respect.

Another effort has been made in SOCO 2014 [18]. SOCO was a challenge defined for detection of source code pairs that are reused. The task was carried out at document level, and in C/C++ and Java. Two datasets were provided: train and

test. Train dataset was labeled and used to train an algorithm that can find source code pairs in which one pair is developed reusing the other one. The test dataset was used to evaluate the accuracy of the developed algorithm with respect to recall, precision, and F1⁴. SOCO contains only 259 Java files and 79 C files, and these examples do not represent realistic software projects (the origin of the source code is unclear).

VI. THREATS TO VALIDITY AND LIMITATIONS

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

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

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

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

VII. CONCLUSIONS AND FUTURE WORK

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

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

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

³Available at <https://sites.google.com/site/asegsecold/projects/seclone> (March 2018).

⁴The F1 score is a measurement provided by the harmonic mean between precision and recall.

REFERENCES

- [1] 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)*, pp. 1–9, IEEE, 2010.
- [2] 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*, vol. 74, no. 7, pp. 470 – 495, 2009.
- [3] P. Weissgerber and S. Diehl, "Identifying refactorings from source-code changes," in *21st IEEE/ACM International Conference on Automated Software Engineering (ASE 2006)*, pp. 231–240, IEEE, 2006.
- [4] S. Kawaguchi, T. Yamashina, H. Uwano, K. Fushida, Y. Kamei, M. Nagura, and H. Iida, "Shinobi: A tool for automatic code clone detection in the ide," in *2009 16th Working Conference on Reverse Engineering*, pp. 313–314, Oct 2009.
- [5] 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*, p. 26, 2008.
- [6] D. Rattan, R. Bhatia, and M. Singh, "Software clone detection: A systematic review," *Information and Software Technology*, vol. 55, no. 7, pp. 1165 – 1199, 2013.
- [7] M. White, M. Tufano, C. Vendome, and D. Poshyanyk, "Deep learning code fragments for code clone detection," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016*, (New York, NY, USA), pp. 87–98, ACM, 2016.
- [8] V. Saini, F. Farmahinifarahani, Y. Lu, P. Baldi, and C. Lopes, "Oreo: Detection of clones in the twilight zone," in *Proceedings of the 2018 26th ACM SIGSOFT International Symposium on Foundations of Software Engineering (To Appear)*, FSE 2018, (New York, NY, USA), ACM, 2018. <https://arxiv.org/abs/1806.05837>.
- [9] Z. Li, S. Lu, S. Myagmar, and Y. Zhou, "Cp-miner: finding copy-paste and related bugs in large-scale software code," *IEEE Transactions on Software Engineering*, vol. 32, pp. 176–192, March 2006.
- [10] H. Sajjani, V. Saini, J. Svajlenko, C. K. Roy, and C. V. Lopes, "Sourcercc: Scaling code clone detection to big-code," in *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, pp. 1157–1168, May 2016.
- [11] J. Svajlenko and C. K. Roy, "Fast and flexible large-scale clone detection with cloneworks," in *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*, pp. 27–30, May 2017.
- [12] L. Jiang, G. Misherggi, 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 (ICSE 2007)*, pp. 96–105, IEEE Computer Society, 2007.
- [13] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier, "Clone detection using abstract syntax trees," in *Proceedings. International Conference on Software Maintenance (Cat. No. 98CB36272)*, pp. 368–377, Nov 1998.
- [14] M. Gabel, L. Jiang, and Z. Su, "Scalable detection of semantic clones," in *Proceedings of the 30th International Conference on Software Engineering, ICSE '08*, (New York, NY, USA), pp. 321–330, ACM, 2008.
- [15] C. K. Roy and J. R. Cordy, "Nicad: Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization," in *2008 16th IEEE International Conference on Program Comprehension*, pp. 172–181, June 2008.
- [16] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, and E. Merlo, "Comparison and evaluation of clone detection tools," *IEEE Transactions on Software Engineering*, vol. 33, pp. 577–591, Sept 2007.
- [17] H. Murakami, Y. Higo, and S. Kusumoto, "A dataset of clone references with gaps," in *Proceedings of the 11th Working Conference on Mining Software Repositories, MSR 2014*, (New York, NY, USA), pp. 412–415, ACM, 2014.
- [18] E. Flores, P. Rosso, L. Moreno, and E. Villatoro-Tello, "On the detection of source code re-use," in *Proceedings of the Forum for Information Retrieval Evaluation, FIRE '14*, (New York, NY, USA), pp. 21–30, ACM, 2015.
- [19] 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*, pp. 476–480, Sept 2014.
- [20] 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)*, pp. 596–600, Oct 2016.
- [21] 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*, (New York, NY, USA), pp. 1066–1077, ACM, 2018.
- [22] J. Svajlenko and C. K. Roy, "Evaluating clone detection tools with bigclonebench," in *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pp. 131–140, Sept 2015.
- [23] V. Saini, H. Sajjani, and C. Lopes, "Cloned and non-cloned java methods: a comparative study," *Empirical Software Engineering*, vol. 23, pp. 2232–2278, Aug 2018.
- [24] V. Saini, F. Farmahinifarahani, Y. Lu, P. Baldi, and C. Lopes, "Mondego/oreo-artifact: Oreo first release," July 2018.
- [25] P. Baldi and Y. Chauvin, "Neural networks for fingerprint recognition," *Neural Computation*, vol. 5, no. 3, pp. 402–418, 1993.
- [26] N. Göde and R. Koschke, "Incremental clone detection," in *13th European Conference on Software Maintenance and Reengineering (CSMR)*, pp. 219–228, IEEE, 2009.
- [27] 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)*, pp. 236–238, IEEE, 2013.
- [28] "Copy/paste detector (cpd)." https://pmd.github.io/pmd-6.6.0/pmd_userdocs_cpd.html. Accessed: 2018-08-23.