Threshold-free Code Clone Detection for a Large-scale Heterogeneous Java Repository

Iman Keivanloo
Department of Electrical and Computer Engineering
Queen's University
Kingston, Ontario, Canada
iman.keivanloo@queensu.ca

Feng Zhang
School of Computing
Queen's University
Kingston, Ontario, Canada
feng@cs.queensu.ca

Ying Zou
Department of Electrical and Computer Engineering
Queen's University
Kingston, Ontario, Canada
ying.zou@queensu.ca

***Abstract*—Code clones are unavoidable entities in software ecosystems. A variety of clone-detection algorithms are available for finding code clones. For Type-3 clone detection at method granularity (*i.e.*, similar methods with changes in statements), dissimilarity threshold is one of the possible configuration parameters. Existing approaches use a single threshold to detect Type-3 clones across a repository. However, our study shows that to detect Type-3 clones at method granularity on a large-scale heterogeneous repository, multiple thresholds are often required. We find that the performance of clone detection improves if selecting different thresholds for various groups of clones in a heterogeneous repository (*i.e.*, various applications). In this paper, we propose a threshold-free approach to detect Type-3 clones at method granularity across a large number of applications. Our approach uses an unsupervised learning algorithm, *i.e.*, k-means, to determine true and false clones. We use a clone benchmark with 330,840 tagged clones from 24,824 open source Java projects for our study. We observe that our approach improves the performance significantly by 12% in terms of F-measure. Furthermore, our threshold-free approach eliminates the concern of practitioners about possible misconfiguration of Type-3 clone detection tools.**

Keywords*—*clone detection; clone search; clustering; unsupervised learning; large-scale repository

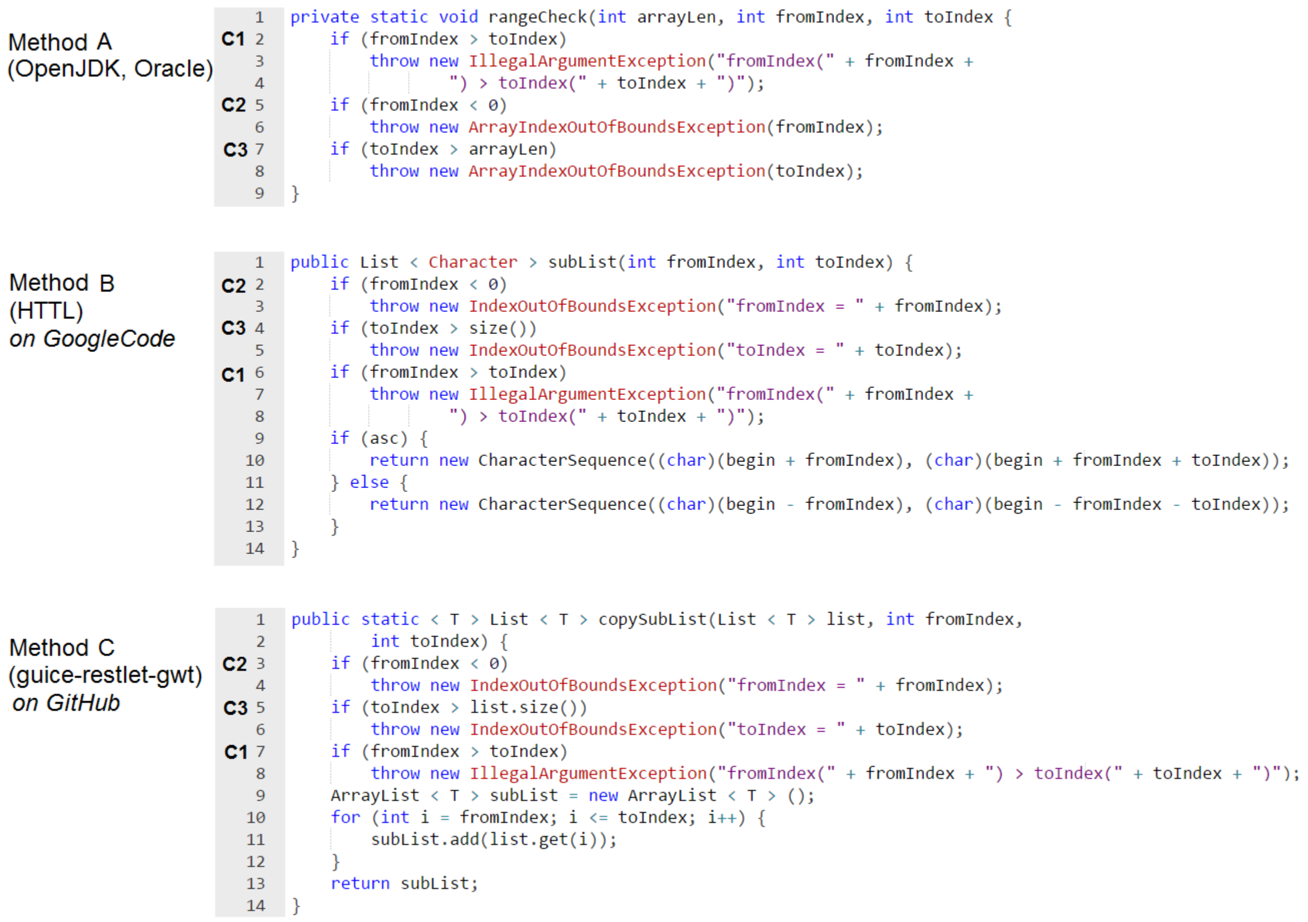
I. Introduction

Code clones commonly exist in software systems. Type-1 clones are made of exactly the same code fragments regardless of the presentation style, comments or white spaces [1]. Further changes of token values (*e.g.*, variable names) lead to Type-2 clones. Type-3 clones are modified copies of the original code with statement changes (*e.g.*, deletion) [22]. Clone detection is a well-established research discipline since the 1970s [26]. Emerging applications of clone detection usually find clones of a specific functionality (*e.g.*, bubble sort) across a large-scale heterogeneous source-code repository (*e.g.*, open source projects on SourceForge, GitHub, or GoogleCode). For example, Ossher *et al.* [25] and Schwarz *et al.* [31] use clone detection to determine the prevalence of reuse within open source development community. Inoue *et al.* [30] use large-scale clone detection for studying the evolution of cloned files across similar software systems. Koschke [21] studies the application of large-scale inter-system clone detection for license infringement detection. Higo *et al.* [11] observe that new programming libraries can be identified using clone detection across a large number of software systems. Furthermore, Type-3 clone detection can be used to intelligently tag code fragments [28], or recommend code examples [19].

Ishihara *et al.* [14] argue that detecting cloned methods with similar functionality across a large number of software systems is one of the basic requirements of such emerging applications. To detect up to Type-3 clones at method granularity, the similarity degree between two candidate methods should be measured. For example, NiCad [29] uses the longest common subsequence; Deckard [16] uses Euclidean distance. Common to these approaches [6] [16] [17] [20] [29] [34], the precision is maintained at an acceptable level by excluding any candidate clone-pair with low similarity degree from the final clone set. A cutoff value on the similarity degree is therefore required to determine how much dissimilarity (between two methods) is allowed for a clone-pair. This dissimilarity threshold constitutes the main configuration parameter of such Type-3 clone detection algorithms at method granularity.

We observed that there are two major challenges associated with the existing threshold-based approach. First, finding the proper value of threshold is an essential step for each proposed clone detection algorithm. Changing the value of the threshold from low to high considerably increases precision and decreases recall in Type-3 clone detection, *e.g.*, [34]. Wang *et al.* [35] report that 61% out of 185 studies using clone detection algorithms explicitly mention the configuration choice as a threat to validity. The default thresholds, as discussed by Wang *et al.* [35], can also be biased towards a specific observation, experiment, or dataset. This problem puts limitation on the emerging applications (*e.g.*, [17]) that depend on Type-3 clone detection algorithms. Second, existing approaches use a single threshold to detect Type-3 clones across a source-code repository. However, we observe that the performance improves if multiple thresholds are used to detect clones across a large-scale heterogeneous source-code repository. For example, different thresholds are preferred to detect clones related to "bubble sort" and "Zip file decompression" functionalities across Java open source systems. A potential explanation for this phenomenon is that methods implementing bubble sort are exposed to less modifications and diversity than methods implementing zip file decompression.

In response to the two aforementioned challenges, we propose a solution that does not require a threshold (as part of the configuration) for Type-3 clone detection. We use an



```java
private static void rangeCheck(int arrayLen, int fromIndex, int toIndex {
    if (fromIndex > toIndex)
        throw new IllegalArgumentException("fromIndex(" + fromIndex +
                    ") > toIndex(" + toIndex + ")");
    if (fromIndex < 0)
        throw new ArrayIndexOutOfBoundsException(fromIndex);
    if (toIndex > arrayLen)
        throw new ArrayIndexOutOfBoundsException(toIndex);
}
```

```java
public List < Character > subList(int fromIndex, int toIndex) {
    if (fromIndex < 0)
        throw new IndexOutOfBoundsException("fromIndex = " + fromIndex);
    if (toIndex > size())
        throw new IndexOutOfBoundsException("toIndex = " + toIndex);
    if (fromIndex > toIndex)
        throw new IllegalArgumentException("fromIndex(" + fromIndex +
                    ") > toIndex(" + toIndex + ")");
    if (asc) {
        return new CharacterSequence((char)(begin + fromIndex), (char)(begin + fromIndex + toIndex));
    } else {
        return new CharacterSequence((char)(begin - fromIndex), (char)(begin - fromIndex - toIndex));
    }
}
```

```java
public static < T > List < T > copySubList(List < T > list, int fromIndex,
        int toIndex) {
    if (fromIndex < 0)
        throw new IndexOutOfBoundsException("fromIndex = " + fromIndex);
    if (toIndex > list.size())
        throw new IndexOutOfBoundsException("toIndex = " + toIndex);
    if (fromIndex > toIndex)
        throw new IllegalArgumentException("fromIndex(" + fromIndex + ") > toIndex(" + toIndex + ")");
    ArrayList < T > subList = new ArrayList < T > ();
    for (int i = fromIndex; i <= toIndex; i++) {
        subList.add(list.get(i));
    }
    return subList;
}
```

Fig. 1: The 9-line method (A) discussed in the trial case of Oracle and Google. Methods B and C are two Type-3 clones of method A that we identified in projects hosted on GitHub and GoogleCode.

unsupervised learning step to filter the candidate pairs that are probably not clones. Specifically, we apply k-means clustering to replace the threshold-based cutoff step in the clone detection process. However, we have to determine the number of expected clusters as part of the configuration of k-means. To automatically identify the proper number of clusters at run-time for each group of candidate clones, our approach uses the Friedman quality optimization method [7] that quantifies the quality of clustering by evaluating the inter and intra-cluster distance. The optimization method maximizes the distance between clusters while minimizing the inter-cluster distance.

In this paper, we studied the following three research questions:

RQ1: *Do we need more than a single threshold for detecting clones across a source code repository?* In our analysis, we use a clone benchmark created using a source-code repository of 24,824 Java open source projects [33]. We observe that the best threshold for Type-3 clone detection varies for different functionalities when finding similar methods across a large number of software systems. The observation suggests that thresholds should be selected dynamically for Type-3 clone detection on such a large-scale repository.

RQ2: *Can we improve the performance of detecting Type-3 clones at method granularity using multiple thresholds?* We assume that an ideal multi-threshold approach exists. The ideal approach is the one that always selects the best threshold at run-time. We show that a multi-threshold approach indeed improves the performance of Type-3 clone detection. It is worth finding a practical multi-threshold solution like our threshold-free approach.

RQ3: *Does our threshold-free approach outperform threshold-based clone detection?* We propose a k-means based approach for clustering true and false clones which is called threshold-free clone detection. We observe that our approach can improve the performance by 12% in terms of F-measure.

The remainder of this paper is organized as follows. Section II summarizes the motivation of our research. Our approach is described in Section III. Sections IV and V present the design and results of our case study. Sections VI and VII review threats to validity and related work. Finally, we conclude and provide insights for future work in Section VIII.

II. Motivation

In this section, we first review a motivating application for detecting code clones in a large-scale heterogeneous repository. We then review an example that motivates our threshold-free clone detection.

A. *Motivating application*

A 9-line method (*i.e.*, method A in Figure 1) responsible to implement "range check" functionality for an array data

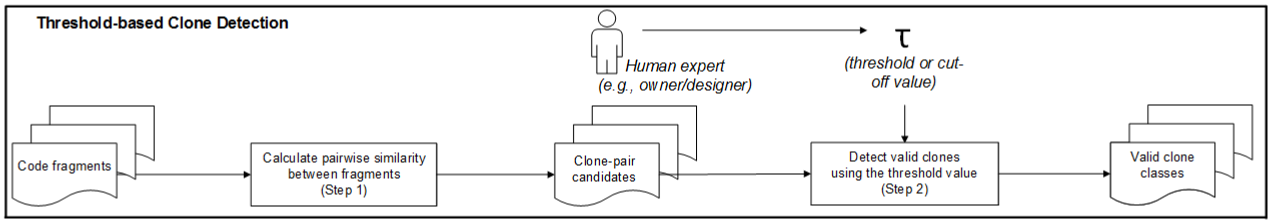

Threshold-based Clone Detection
Human expert
(e.g., owner/designer)
τ
(threshold or cut-off value)
Code fragments
Calculate pairwise similarity between fragments (Step 1)
Clone-pair candidates
Detect valid clones using the threshold value (Step 2)
Valid clone classes

Fig. 2: Overview of threshold-based Type-3 clone detection at method granularity.

structure is discussed in the trial case of Oracle and Google[1]. In this case, Oracle claims that the 9-line method used by Google has an incompatible license. The method is made of three "if" statements. Since implementation of the range check functionality using these if statements is an intuitive approach, an interesting question to ask is the following:

"Is Google's Android the only open source software system that adopted the 9-line method from Oracle's systems with an incompatible license?".

Answering such questions helps open source developers to avoid inadvertent license infringement like the case between Oracle and Google. In general, we require to identify clones of a method which implements a specific functionality on a large-scale heterogeneous repository, *e.g.*, [14]. A corpus of open source software systems crawled from the online resources (*e.g.*, SourceForge, GitHub, and GoogleCode) is an example of a large-scale heterogeneous repository [37]. The major challenge is to detect such clones with an acceptable balance between recall and precision. Recall is important since missing a true clone-pair can lead to legal cases. Similarly, precision plays an important role as well [21]. Low precision causes wasting of human resources since the outcome of the clone detection step has to be manually verified by experts [21].

B. Background - threshold-based Type-3 clone detection

Figure 1 shows two methods (*i.e.*, methods B and C) that are Type-3 clones of Oracle's 9-line method (*i.e.*, method A). Method B[2] in Figure 1 is part of the HTTL open source project that is a hyper-text template language and engine hosted on GoogleCode. Method C[3] in Figure 1 is part of a dependency injection framework (guice-restlet-gwt) hosted on GitHub.

For detecting such similar methods, we require a scalable approach that can detect Type-3 clones at method granularity. Although both methods B and C in Figure 1 use a similar implementation as method A for the range check functionality, they include additional source code for other purposes such as "for loop" statements for partially copying of a list (line 10, method C). Furthermore, the condition C1 in method A is moved to the middle of the code in methods B and C. In addition, there exist fine-grained dissimilarities among the three snippets. For example, method A retrieves the size of the array from a parameter called arrayLen at line 7, while the other two fetch the size via a method call at line 4 (method B) and line 5 (method C) in Figure 1.

[1] http://en.wikipedia.org/wiki/Oracle_v._Google (The trial case of Oracle v.s. Google.)

[2] https://code.google.com/p/httl/source/browse/trunk/httl/src/main/java/com/googlecode/httl/support/sequences/CharacterSequence.java?r=16#155

[3] https://github.com/nightscape/guice-restlet-gwt/blob/master/src/main/java/org/restlet/engine/util/ListUtils.java#L59

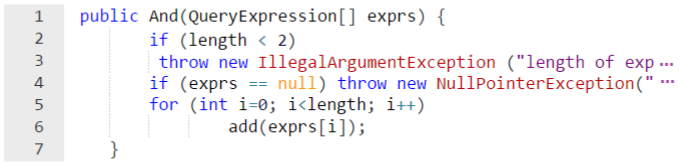

```
1  public And(QueryExpression[] exprs) {
2          if (length < 2)
3           throw new IllegalArgumentException ("length of exp…
4          if (exprs == null) throw new NullPointerException(" …
5          for (int i=0; i<length; i++)
6                  add(exprs[i]);
7      }
```

Fig. 3: A candidate clone of method A in Figure 1. The low lexical similarity between the two methods suggests that this is not an actual (true) clone of method A in Figure 1 which implements the range check functionality for arrays. Such clone candidates are referred to as false clones [33].

The major common steps in the existing Type-3 clone detection algorithms are shown in Figure 2. First, a set of candidate clones for a target method (*e.g.*, method A in Figure 1) are located using an indexing mechanism (*e.g.*, [18] [8] [12] [21] [30]). In this example, methods B and C would be probably detected as candidate clones of method A in Figure 1. We denote the set of candidate clones as J_{m_i} for the target method m_i. The members of the candidate set J_{m_i} create a set of candidate clone-pairs with the target method m_i. The candidate clone-pairs set is denoted by C_{m_i}. Each *candidate clone-pair* can be represented as a triple (m_i,m_j,ς^{m_i,m_j}) where m_j denotes a *candidate clone* from the candidate clone set J_{m_i} and ς^{m_i,m_j} is the similarity degree between the target method m_i and the candidate clone m_j. Then, each candidate clone-pair in the candidate set C_{m_i} is evaluated against the threshold in Step 2 (Figure 2), *i.e.*, if ($\varsigma^{m_i,m_j} > \tau$) {report ($m_i$,$m_j$) as a clone-pair} else {discard the pair}, where τ denotes the threshold.

Figure 3 shows a candidate clone of Oracle's 9-line range check code (method A in Figure 1). Although the candidate clone (Figure 3) shares some similarity with the target snippet (method A in Figure 1) such as the "if" and "throw" statements in lines 2 and 3, the method clearly is not an actual clone of the range check functionality as implemented in Figure 1. To avoid low precision, we require to detect such false candidates (*e.g.*, Figure 3) from actual candidates (*e.g.*, methods B and C in Figure 1) as clones of method A in Figure 1. To achieve this goal, existing approaches for Type-3 clone detection at method granularity use the threshold-based filtering step to balance the trade off between precision and recall [29] by removing false candidate clones such as Figure 3 from the final result set.

C. Motivation for threshold-free Type-3 clone detection

In this paper, we aim to find an alternative solution to replace the classical threshold-based filtering step (Step 2

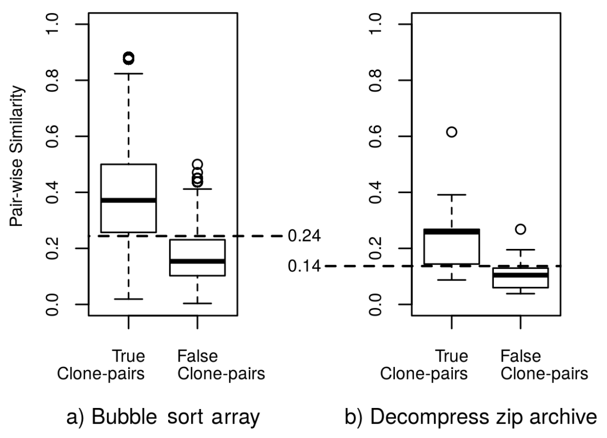


Fig. 4: A motivating example that shows the optimum threshold (marked by the horizontal dashed lines) varies for detecting clones of different functionalities. The observation is based on clones of the corresponding functionalities recorded in a clone benchmark [33] from online resources (*e.g.*, GoogleCode and SourceForge).

Algorithm 1: Using unsupervised learning for threshold-free clone detection

```
Input:
$m_i$: target method, e.g., method A in Figure 1.
$e$: the optimization method for the clustering algorithm.
$d$: the distance function of our approach.
$h$: the inverted index covering all methods in repository.
Output:
methods in the repository that are Type-3 clones of $m_i$.
1  $J_{m_i}$ = findCandidates($m_i$,$h$);
2  $n_{m_i}$ = size($J_{m_i}$);
3  $v_{best} = null$;
4  for $k = 2$ to $n_{m_i}$ do
5      $v = e(k, J_{m_i})$;
6      if $v > v_{best}$ then
7          $k_{best}^{m_i,e} = k$;
8          $v_{best} = v$;
9      end
10 end
11 $clusters^{m_i,e} = k\text{-}means(k_{best}^{m_i,e}, J_i, d)$;
12 $output^{m_i,e} = dropTail(clusters^{m_i,e})$;
```

shown in Figure 2). There exist two motivations for us to propose a threshold-free Type-3 clone detection for applications discussed in Sections I and II-A. First, to identify the value of the threshold, a sensitive analysis is required (*e.g.*, [29]), which is a time-consuming and error-prone process [35]. The value of the identified threshold might need to be updated for each source-code repository, as discussed by Wang *et al.* [35]. Second, a single threshold might lead to poor performance of detecting Type-3 clones in a large-scale repository. Figure 4 shows pair-wise similarity for true and false clones of two sample functionalities, *i.e.*, bubble sort and zip file decompression. The data are extracted from a benchmark [33] of code clones that exist in open source projects from SourceForge and GoogleCode. *True clones* are Type-3 clone-pairs that both methods implement the same corresponding functionality. *False clones* are pairs that share some degree of similarity but they are not a true clone-pair of the corresponding functionality, *e.g.*, method A in Figure 1 and the method in Figure 3. As shown in Figure 4, the observation suggests that to detect clones related to bubble sort functionality, a threshold about 0.24 is preferred (marked by the horizontal dashed line). According to the benchmark, this threshold achieves the best balance between precision and recall measured by F-measure. The recall decreases if we move the threshold toward the top of the figure. Similarly, the precision decreases if we move the threshold toward the bottom of the figure. However, for the decompression functionality, a threshold about 0.14 is preferred. If we use a single-threshold for both cases, either the precision or recall of one of the two cases will be negatively affected. This motivating example suggests that different thresholds for various functionalities should be considered to properly balance precision and recall.

In this paper, we propose a threshold-free Type-3 clone detection approach for applications discussed in Sections I and II-A. First, a threshold-free clone detection approach improves the ease of using clone detection tools. It does not require the users to find the optimized threshold as part of the configuration step. Instead, it automatically infers the required threshold from the data. Second, it improves the performance (*i.e.*, F-measure) by dynamically inferring the threshold for various functionalities.

III. Our Approach for Threshold-free Clone Detection

Preparation. Algorithm 1 illustrates the details of our approach which replaces Step 2 in the traditional threshold-based approach shown in Figure 2. First, our approach identifies the set of candidate methods for the target method m_i using an inverted index implemented by a hash table (*e.g.*, [18] and [12]). The inverted index helps us to identify the set of candidate clones in constant time [18]. Second, the distance between candidates of the target method m_i is calculated. The distance is denoted by D^{m_i,m_j} where m_j is a candidate clone of the target method m_i. The distance between two methods is measured based on NiCad's Unique Percentage of Items (UPI) [29] which uses the number of unique cases that do not appear in the longest common subsequence (LCS) [13] of methods m_i and m_j. In our approach, each candidate is represented using its distance to the target method. The relative $distance$ between two candidate methods of m_i is $d(m_a, m_b) = D^{m_i,m_a} - D^{m_i,m_b}$.

Clustering. We apply a clustering algorithm on the candidate methods represented by their distance to the target method. A clustering technique partitions data into clusters of similar elements [2]. The elements inside each cluster are similar to each other (*i.e.*, cohesiveness) and dissimilar to the elements of other clusters (*i.e.*, decoupled). We use the distance method described in the preparation step to quantify dissimilarity between the entities. One of the common clustering methods is k-means [10]. k-means works well for numerical attributes from a geometrical and statistical perspective [2]. Our approach uses k-means clustering to achieve threshold-free clone detection. As an unsupervised learning method, k-means has a configuration parameter k which specifies the number of expected clusters. The algorithm assigns each input data point to one of the k clusters by considering two clustering criteria, decoupling and cohesiveness of clusters. Therefore, the selected value for k affects the outcome of the algorithm.

Optimization. A proper value for k leads to a high quality clusters where clusters are highly distant from each other and objects of each cluster are highly close to each other. Optimization methods for a clustering algorithm evaluate the resulting clusters from cohesiveness and decoupling points of view [24]. To identify the proper value for k, we choose to use the Friedman quality optimization method [7]. Friedman method can explore the relation of data points for non-hierarchical clustering algorithms (*e.g.*, k-means). Friedman method uses the following criterion to quantify the quality of clustering results.

$$Friedman = trace(W^{-1}B) \tag{1}$$

where B is the between-group scatter matrix and W is the within-group scatter matrix. Scatter matrices quantify separability within and between groups. The detailed definition of B and W can be found in the work by Friedman and Rubin [7].

To find the best number of clusters, we evaluate k-means performance for a set of possible k. For each k, we compute the $Friedman$ criterion. The maximum value observed by the method (denoted by v in Algorithm 1 - line 5) identifies the optimized k for k-means clustering.

Threshold-free detection. In the algorithm, n_{m_i} denotes the total number of candidate methods reported by the inverted index (Line 1) for the target method m_i. When our algorithm identifies the best value for the number of clusters k (*i.e.*, $k_{best}^{m_i,e}$) for the target method m_i from the optimization method point of view, it passes the value to k-means to generate the clusters of the candidates. In the last step (*i.e.*, drop tail), amongst the $k_{best}^{m_i,c}$ clusters generated by k-means, our approach drops the cluster where the centroid has the largest distance from the target method m_i. The underlying rationale is that the cluster with the largest distance from the target method m_i has the highest probability to contain false clones. This heuristic is motivated by our earlier experience in evaluating similarity scores of 32K distinct clone-pairs. We also noticed a similar observation reported in an earlier study [9] that false clones "are considerably farther apart than those that are considered potential clones". Finally, the members in the remaining clusters are reported as true clones of method m_i.

IV. Case Study Setup

We perform a case study with three research questions to evaluate the feasibility and performance of our proposed approach. In this section, we present the setup of our study.

A. Data collection

Source code repository. We use IJaDataset 2.0 as the source-code repository [19]. This dataset covers 24,824 projects downloaded from SVN, Git, and CVS repositories published on SourceForge, GoogleCode, and GitHub. Table I summarizes the source-code repository.

Clone benchmark. To measure the clone detection performance, we require a gold dataset (*e.g.*, [1]) that specifies true and false candidate clones that might exist in the source-code repository. Specifically, we require a large-scale benchmark that includes clones across software systems. Svajlenko *et al.* [33] created a clone benchmark called BigCloneBench which is publicly available[4]. The benchmark consists of clones of specific functionalities in IJaDataset, *i.e.*, the same source-code repository that we use in this paper. The benchmark is created by mining IJaDataset for methods that implement a set of candidate functionalities. The benchmark is initially released with 59,688 tagged candidate clones [33], and later extended via an incremental process. We acquired an extended version of the benchmark which was available at the time of our analysis from the benchmark maintainer. Table II provides a summary of the extended version of the benchmark used in our analysis. This snapshot includes 330,840 candidate clones that are tagged by the benchmark curators for a set of functionalities.

TABLE I: Summary of the source-code repository (IJaDataset 2.0).

Feature	Value
Java projects	24,824
Java files	2,882,458
LOC	300 Million
Snippets	23.7 Million

For each functionality, a set of candidate clones that might implement the functionality are reported in the benchmark. For example, 14,054 candidate clones are identified in the benchmark (Table II) for the "Binary Search" functionality from the source code of 24,824 Java projects in IJaDataset (Table I). Furthermore, the candidate clones for the functionality are labeled as either $true\ clone$ or $false\ clone$ in the benchmark. A $true\ clone$ is a method that indeed implements a functionality in the benchmark [33]. A $false\ clone$ is a method that does not implement the functionality. For example, as identified by the benchmark, there exist 1,758 true clones for "Binary Search" in the source-code repository (*i.e.*, IJaDataset 2.0). For the same functionality, 12,296 clones exist in the source-code repository that seem to implement the "Binary Search" functionality but they do not. In total, there exist 43,085 true clones for the covered functionalities and 287,755 snippets that are false clones tagged by the benchmark curators.

B. Performance evaluation measure

We measure the performance of a clone detection experiment for each functionality using the information provided in the benchmark (*i.e.*, candidate snippets, true, and false clones). We calculate F-measure as defined in Equation (2) to observe the overall performance with regard to precision (3) and recall (4).

$$F-measure_g,\tau = 2 \times \frac{Precision_g,\tau \times Recall_g,\tau}{Precision_g,\tau + Recall_g,\tau} \tag{2}$$

$$Precision_g,\tau = \frac{B_g \cap D_g,\tau}{D_g,\tau} \tag{3}$$

[4]http://github.com/clonebench/BigCloneBench (the clone benchmark by Svajlenko *et al.* [33] for IJaDataset 2.0).

TABLE II: The summary of the clone benchmark used in our analysis.

Id	Functionality	Number of candidates	Number of true clones	Number of false clones	min LOC	max LOC	mean LOC	median LOC	std.dev LOC	var LOC
1	Binary Search	14054	1758	12296	3	5118	53	26	161	25796
2	Bubble Sort Array	18291	2303	15988	5	2232	43	21	98	9661
3	Call Method Using Reflection	2170	1666	504	3	2935	64	26	233	54076
4	Connect to Database	415	409	6	4	575	37	24	50	2542
5	Copy File	222626	18518	204108	2	4682	30	16	93	8612
6	Get Prime Factors	194	42	152	3	568	69	25	139	19277
7	Create Encryption Key Files	507	57	450	5	194	47	33	39	1546
8	Decompress Zip Archive	27	15	12	4	126	50	41	39	1484
9	Delete Folder and Contents	695	549	146	7	772	29	15	52	2735
10	Download From Web	41571	2733	38838	3	1557	31	22	41	1693
11	Connect to FTP Server	6589	2398	4191	4	251	35	26	29	815
12	Convert Date String Format	212	58	154	3	865	75	30	115	13169
13	Copy Directory	667	301	366	8	1162	67	35	99	9788
14	CRC32 File Checksum	318	282	36	5	6131	50	18	344	118136
15	Encrypt To File	180	74	106	7	610	48	30	69	4820
16	Execute External Process	1001	929	72	6	647	53	29	55	2972
17	Execute Update and Rollback	3313	1503	1810	4	1288	45	32	60	3578
18	Extract Matches Using Regex	505	502	3	7	500	36	22	42	1771
19	Fibonacci	211	211	0	5	12	7	7	1	1
20	File Dialog	1908	1908	0	5	507	41	20	65	4271
21	Get MAC Address String	47	43	4	11	78	31	27	15	233
22	Initialize Java Eclipse Project	22	22	0	28	243	81	54	70	4862
23	Instantiate Using Reflection	905	861	44	3	2935	44	24	144	20674
24	Load Custom Font	96	75	21	5	397	49	24	73	5369
25	Load File into Byte Array	360	158	202	4	2431	52	22	149	22153
26	Open File in Desktop Application	281	104	177	3	415	39	20	57	3199
27	Open URL in System Browser	432	387	45	3	415	31	16	46	2099
28	Parse CSV File	250	201	49	5	305	65	57	43	1888
29	Parse XML to DOM	495	391	104	3	450	46	27	62	3878
30	Play Sound	234	111	123	3	6131	190	28	718	516033
31	Resize Array	462	439	23	4	434	22	12	38	1454
32	Secure Hash	5906	1342	4564	3	918	24	16	33	1080
33	Send E-Mail	272	239	33	3	263	46	34	38	1419
34	Setup SGV Event Handler	1282	10	1272	2	255	16	10	18	331
35	Setup SGV	101	23	78	2	152	22	13	24	560
36	Shuffle Array in Place	846	234	612	3	181	32	14	37	1393
37	Take Screenshot to File	400	104	296	3	1217	32	17	69	4772
38	Transpose a Matrix	592	542	50	6	1574	39	17	78	6126
39	Write PDF File	287	158	129	10	2431	79	37	195	37937
40	Zip Files	2116	1425	691	3	574	35	24	42	1797

$$Recall_g, \tau = \frac{B_g \cap D_g, \tau}{B_g} \tag{4}$$

where g denotes one of the functionalities in the benchmark. The threshold used for Type-3 clone detection is identified by τ. B_g is the set of all true clones related to functionality g in the benchmark. D_g, τ is the set of clones detected by the tool as the answer for functionality g using threshold τ.

C. Threshold-based clone detection

In our case study, we use NiCad's similarity measure [29]. NiCad is a clone detector for Type-3 clone detection at method granularity. NiCad uses the length of the longest common subsequence between a target fragment and its candidate fragments as the similarity measure. It depends on a distance measure named Unique Percentage of Items (UPI) to distinguish clone-pairs and non clone-pairs. We consider nine possible thresholds of NiCad in our analysis. The nine possible thresholds of NiCad are identified as T #1 to #9.

V. Case Study Results

This section presents and discusses the results of our three research questions. For each research question, we present the motivation behind the question, the analysis approach and a discussion on our findings.

***RQ1:* Do we need more than a single threshold for detecting clones across a source code repository?**

Motivation. Existing approaches use a single threshold to distinguish true clones from false clones within a source-code repository. The value of the threshold is determined using sensitive analysis or search-based optimization (*e.g.*, [35]). The threshold that achieves the best overall performance is used for clone detection on the complete repository (*e.g.*, [29]). However, we conjecture that a single threshold might not always lead to the best clone detection performance for various functionalities in a repository, as discussed in our motivating example in Section II-C and Figure 4. In this exploratory research question, we investigate if a single threshold can always achieve the best clone detection performance on a source-code repository. The analysis of **RQ1** is our first step towards understanding if threshold-free clone detection is necessary.

Approach. In our experiments, we consider IJaDataset 2.0 as the source-code repository. We use NiCad to detect clones in the source-code repository. NiCad supports nine thresholds for Type-3 clone detection. As an exploratory study, we consider all of the nine possible thresholds in our analysis.

As described in Section IV-B, we measure the performance of each threshold per functionality using the benchmark [33]. To observe if a single threshold can achieve a superior per-

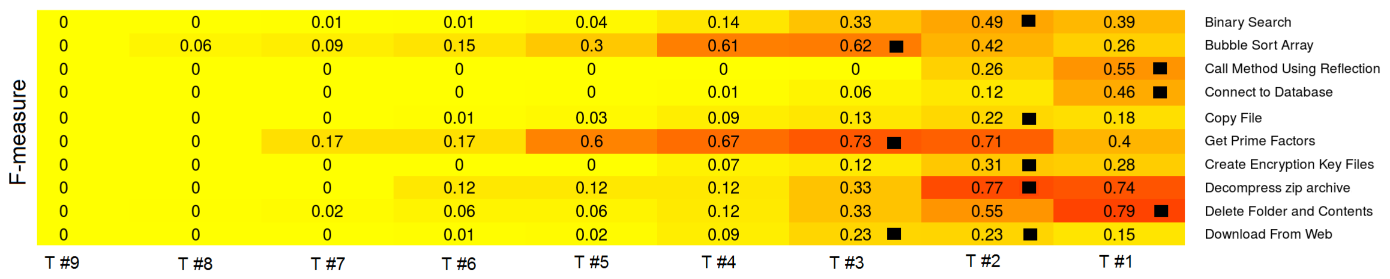


Fig. 5: The F-measure values that are achieved for the functionalities. The performance achieved by each threshold (#1 to #9) is reported separately. The best threshold for each functionality is marked by a pin.

formance for various functionalities, we also report the best threshold for each functionality that exists in the benchmark (Section IV-A). The best threshold is the one that leads to the best F-measure (Section IV-B) for detecting clones that belong to a specific functionality.

Findings. Figure 5 shows the performance in terms of F-measure achieved for the studied functionalities in the benchmark (Table II). Due to lack of the space, we show only ten sample functionalities from the benchmark in Figure 5. For each functionality, the threshold that leads to the highest F-measure is tagged in the heat map. As we can observe, the performance varies as we use different thresholds for each functionality. The darker the color is, the higher performance is achieved for the corresponding functionality. For example, in detecting clones related to "Binary Search", the best F-measure is achieved when we use threshold #2. However, we can observe that threshold #2 is not consistently the best choice for clone detection in other functionalities. For example, the best performance for "Bubble Sort" is achieved using threshold #3, as highlighted in Figure 5.

A single threshold cannot achieve the best F-measure across various functionalities for Type-3 clone detection on a large-scale heterogenous repository.

***RQ2:* Can we improve the performance of detecting Type-3 clones at method granularity using multiple thresholds?**

Motivation. In **RQ1**, we observe that the best thresholds for different functionalities in a single large-scale repository are different. Such observation motivates us to investigate if using various thresholds for different functionalities (*i.e.*, clone groups) can significantly improve the performance of Type-3 clone detection (measured by F-measure). We refer to this approach as multi-threshold clone detection. Our threshold-free approach is a concrete form of multi-threshold clone detections. Hence, it is essential to examine the usefulness of multi-threshold approach. If we do not observe a significant improvement of multi-threshold approach comparing to the traditional approach (*i.e.*, single-threshold Type-3 clone detection), there is no reason to persuade the idea of threshold-free clone detection.

Approach. To answer this research question, we assume that the ideal multi-threshold detection approach exists. The ideal approach is the one that always selects the best possible threshold from the nine possible thresholds of NiCad for each functionality. In our analysis, the ideal approach uses the best possible thresholds identified in **RQ1** for each functionality. We measure the overall performance using F-measure which is discussed in Section IV-B. We consider all of the nine possible thresholds for Type-3 clone detection using the traditional approach. To compare the performance of multi-threshold approach with the traditional approach, we test the following null hypothesis:

H_0^1: *the ideal multi-threshold clone detection approach does not improve the clone detection performance.*

Hypothesis H_0^1 is two-tailed since it investigates if using multiple thresholds achieves better performance than the single-threshold approach or not. We perform a Mann-Whitney U test [32] for H_0^1 using p-value < 0.05. The Mann-Whitney U test is a non-parametric statistical test to assess whether two independent distributions have equally large values. Non-parametric statistical methods make no assumptions about the distributions of assessed variables. If there is a statistically significant difference (*i.e.*, p-value is less than 0.05), we reject the null hypothesis and conclude that the ideal multi-threshold detection (if exists) outperforms single-threshold Type-3 clone detection approach.

TABLE III: Mapping Cliff's delta to Cohen's standards

Cliff's delta	% of Non-overlap	Cohen's d	Cohen's standards
0.147	14.7%	0.20	small
0.330	33.0%	0.50	medium
0.474	47.4%	0.80	large

To quantify the impact of the observed improvement, we calculate Cliff's delta as the effect size [15]. Cliff's delta estimates non-parametric effect size. It makes no assumption of a particular distribution [15], and is reported to be more robust and reliable than Cohen's d [5]. Cliff's delta represents the degree of overlap between two sample distributions [15]. It ranges from -1 (if all selected values in the first group are larger than the second group) to +1 (if all selected values in the first group are smaller than the second group). It is zero when two sample distributions are identical [3]. Cohen's standards (*i.e.*, small, medium, and large) are commonly used to interpret effect size. Therefore, we map the Cliff's delta to Cohen's standards, using the percentage of non-overlap [15]. The mapping between the Cliff's delta and Cohen's standards is shown in Table III. Cohen [4] states that a medium effect size represents a difference likely to be visible to a careful observer,

TABLE IV: Comparison of F-measures between the ideal multi-threshold approach and each of the single-threshold experiments.

Approach	F-measure (mean)	Cliff's delta (effect size)		p-value
Ideal multi-threshold	0.63	NA		NA
Threshold #9	0.00	0.998	(large)	1.82e-12
Threshold #8	0.01	0.998	(large)	1.82e-12
Threshold #7	0.02	0.998	(large)	1.82e-12
Threshold #6	0.03	0.997	(large)	1.82e-12
Threshold #5	0.06	0.993	(large)	1.82e-12
Threshold #4	0.16	0.970	(large)	5.47e-08
Threshold #3	0.28	0.935	(large)	1.19e-07
Threshold #2	0.44	0.872	(large)	3.82e-07
Threshold #1	0.60	0.484	(large)	5.92e-03

while a large effect is noticeably greater than medium.

Findings. Table IV shows the F-measures for Type-3 clone detection achieved by the ideal multi-threshold selection and the nine possible single-threshold experiments. We observe that among the single-threshold experiments, threshold #1 leads to the best overall performance (*i.e.*, with the highest F-measure). We also observe that the ideal multi-threshold approach achieves higher F-measure than all of the single-threshold experiments. The results of Mann-Whitney U tests show that multi-threshold selection significantly outperforms the single-threshold clone detection experiments. Therefore, we can reject the null hypothesis H_0^1. Moreover, the ideal multi-threshold approach leads to a better performance with large effect size measured via Cliff's delta based on the guideline provided in Table III.

Multi-threshold clone detection can significantly improve the performance of Type-3 clone detection with large effect size.

RQ3: Does our threshold-free approach outperform threshold-based clone detection?

Motivation. In **RQ2**, we observed that the performance is significantly improved when using multiple thresholds for a single heterogeneous source-code repository. As an exploratory study, the observation in **RQ2** was based on the assumption that the ideal multi-threshold clone detection exists. This question aims to investigate if our threshold-free Type-3 clone detection, as a practical multi-threshold solution, can still achieve better performance than the traditional approach (*i.e.*, single-threshold based approach).

Approach. To answer this research question, we apply our approach on the source-code repository (Section IV-A) for Type-3 clone detection. We measure the performance using the large-scale clone benchmark (Section IV-A and Table II) as described in Section IV-B. The performance is measured using F-measure. We compare the performance of our threshold-free approach with the traditional single-threshold approach using all of its nine possible thresholds. Finally, we test the following null hypothesis:

H_0^2: *the threshold-free clone detection approach does not improve the clone detection performance.*

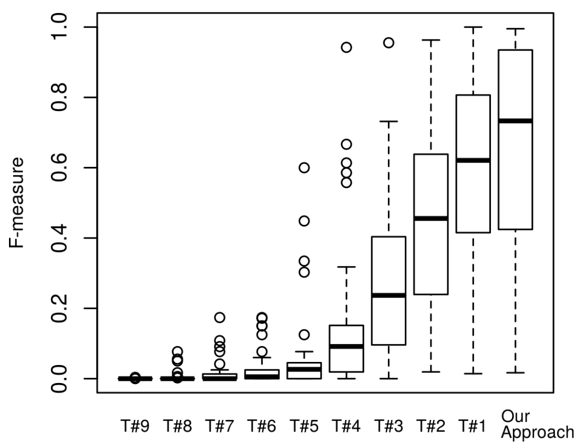


Fig. 6: Summary of the performance evaluation on the proposed threshold-free Type-3 clone detection approach and the traditional single-threshold approach using the clone benchmark. The performance of the single-threshold approach is reported for the nine possible threshold T #1 to T #9.

TABLE V: Comparison of F-measures between our approach (threshold-free) and the traditional single-threshold approach.

Approach	F-measure (mean)	Improvement	Cliff's delta (effect size)		p-value
Threshold-free	0.67	NA	NA		NA
Threshold #9	0.00	<100%	0.989	(large)	1.82e-12
Threshold #8	0.01	<100%	0.988	(large)	1.82e-12
Threshold #7	0.02	<100%	0.985	(large)	1.82e-12
Threshold #6	0.03	<100%	0.982	(large)	1.82e-12
Threshold #5	0.06	<100%	0.962	(large)	7.82e-11
Threshold #4	0.16	<100%	0.904	(large)	6.78e-09
Threshold #3	0.28	<100%	0.828	(large)	3.88e-07
Threshold #2	0.44	52%	0.703	(large)	7.20e-05
Threshold #1	0.60	12%	0.572	(large)	1.52e-03

Hypothesis H_0^2 is two-tailed since it investigates if threshold-free clone detection achieves better or worse performance than the single-threshold approach. We perform a Mann-Whitney U test [32] for H_0^2 using p-value < 0.05. If there is a statistically significant difference (*i.e.*, p-value is less than 0.05), we reject the null hypothesis and conclude that our proposed threshold-free approach outperforms the single-threshold approach for Type-3 clone detection. To quantify the impact of the observed improvement, we further calculate the Cliff's delta effect size.

Findings. Figure 6 presents the boxplot of F-measures achieved by the single-threshold approach for Type-3 clone detection with the nine possible thresholds, and the performance of our approach. We can observe that the threshold-based approach achieves its best performance with threshold #1 (expecting at least 10% similarity), and our approach yields better performance than the single-threshold approach regardless of its chosen threshold. Table V summarizes the result of the statistical test and the effect size when comparing our approach with the traditional approach. In particular, our approach achieves at least 12% improvement (*i.e.*, from 0.60 to 0.67) than the traditional approach. The improvement is statically significant with large effect size. Moreover, the performance of our approach is even higher than the ideal multi-threshold approach that was studied in **RQ2**, since Algorithm-1 makes our approach be able to find more precise thresholds, other than the nine thresholds of the baseline approach.

Threshold-free Type-3 clone detection improves the performance by at least 12% in terms of F-measure, compared to the traditional single-threshold approach.

VI. Threats to Validity

We now discuss the threats to validity of our study following common guidelines provided in [36].

Threats to conclusion validity concern the relation between the treatment and the outcome. Our approach uses k-means as the unsupervised learning approach. Using k-means, we demonstrated the feasibility and advantages of threshold-free clone detection. There might exist a better clustering algorithm for threshold-free clone detection. Further study is encouraged to explore more clustering algorithms.

Threats to internal validity concern our selection of subject systems and analysis methods. To the best of our knowledge, there exists only one clone benchmark for clone detection on large-scale heterogeneous repositories. Furthermore, we acquired an extended snapshot of the benchmark with 330,840 tagged clones from the benchmark maintainer which is four times larger than its initial release [33] in terms of number of covered functionalities. We used NiCad as the baseline in our study since it supports Type-3 clone detection at method granularity.

Threats to external validity concern the possibility to generalize our results. The data used in our study is available at the replication package http://goo.gl/SuUtfi. As part of the replication package, we also released the analysis scripts used in the case study.

VII. Related Work

Earlier studies on clone detection have addressed the scalability challenge for clone detection. Livieri *et al.* [23] proposed a distributed architecture for CCFinder called D-CCFinder that achieves scalability by splitting the repository into subsets. D-CCFinder delegates small size clone detection tasks to client computers, and a master computer records the results reported by clients. Koschke proposes an approach to achieve scalability by using suffix trees [21] as a form of inverted index. Göde and Koschke show that the idea of inverted index using suffix trees can be further extended for scalable incremental clone detection [8] which is released as iClones. Jiang *et al.* propose Deckard [16] which achieves scalability via Locality Sensitive Hashing (LSH). LSH is an extended form of hash-based indexing which finds similar entities with a certain degree of resemblance in constant time complexity. Hummel *et al.*'s research [12] shows that massive scalability for Type-2 clone detection can be achieved using inverted index and MapReduce. Keivanloo *et al.* [18] also studied the application of inverted index for Internet-scale clone detection. Dang *et al.* [6] discuss the applications of clone detection for practitioners. In this paper, we propose an approach that helps practitioners in using scalable Type-3 clone detection algorithms across software systems. Specifically, we focus on improving the performance and the ease of use.

It is challenging to find proper configurations for tools and algorithms in software engineering research. Clone detection research is not an exception in this context; it also suffers from the same problem. The traditional approach is to identify and recommend some default configuration values at the time of proposing a new algorithmic solution (*e.g.*, [29] [16] [34]). However, this approach is usually tedious, and the recommended values are not always generalizable since they can be biased towards a specific observation, experiment, or dataset [35]. Panichella *et al.* [27] show that it is possible to derive the configuration for topic modeling for software engineering applications using a genetic algorithm. Wang *et al.* [35] propose EvaClone which is a search-based solution for configuring clone detection algorithms for empirical studies on software clones. EvaClone is applicable when more than one clone detector is required in a single study. EvaClone derives configuration values that minimize the effect of the confounding factor for result comparison in empirical studies. The derived configuration values are the ones that achieve the highest agreement among the two or more clone detectors. EvaClone is effective in minimizing the effect of the confounding factor when the results of several clone detection algorithms are being compared. However, Wang *et al.* [35] also observe that EvaClone does not necessarily find a superior configuration for applications that require only a single clone detection algorithm. As discussed by Wang *et al.* [35], EvaClone tends to suggest configurations achieving a higher recall and lower precision than the default configuration. In some cases, this behavior leads to a lower F-measure. To the best of our knowledge, EvaClone is the only and closest work to our research. While we propose a solution that eliminates the similarity threshold from the configuration parameters of Type-3 clone detection algorithms, EvaClone can be used to derive the values of any type of configuration parameter. However, EvaClone is designed for applications when more than one clone detection algorithms is required (*e.g.*, empirical studies). Our approach is designed for applications depending on a single clone detection algorithm (*e.g.*, industrial applications [17]). Furthermore, as we showed in **RQ1** and **RQ2**, a single threshold does not lead to a superior performance for large-scale heterogenous repositories. The experiments show that even if EvaClone identifies the best single threshold for the complete repository, our approach can outperform that configuration for Type-3 clone detection clone on a large-scale heterogeneous repository as discussed in **RQ3**.

VIII. Conclusion

In this paper, we show that more than a single threshold is required for Type-3 clone detection at method granularity on a large-scale heterogeneous repository. By considering various thresholds for different functionalities, the performance (measured by F-measure) can be significantly improved. Hence, we propose a threshold-free approach for Type-3 clone detection at method granularity. We apply k-means clustering to separate true and false clones. We use the Friedman method to evaluate the quality of clustering for different k values, so that we can automatically determine the number of expected clusters for k-means clustering. Therefore, our approach is fully automated and threshold free. We show that our approach improves the performance significantly (*i.e.*, 12% increase on F-measure) when applied on a large-scale heterogeneous repository. Since our approach is threshold free, the concern about possible misconfiguration is also eliminated. As the immediate future work, we plan to apply our approach for Type-4 clone detection.

IX. Acknowledgments

The authors would like to thank Dr. Daniel German for motivating us to study the problem of clone detection for cases similar to Google versus Oracle license infringement case. The authors would like to thank Dr. Audris Mockus for his invaluable feedback and comments in the early stages of our research. The authors would like to thank the anonymous reviewers for their invaluable comments.

References

[1] 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, no. 9, pp. 577–591, 2007.

[2] P. Berkhin, "A survey of clustering data mining techniques," in *Grouping Multidimensional Data*, J. Kogan, C. Nicholas, and M. Teboulle, Eds. Springer Berlin Heidelberg, 2006, pp. 25–71.

[3] N. Cliff, "Dominance statistics: Ordinal analysis to answer ordinal questions," 1993.

[4] Cohen, "A power primer." Psychological Bulletin, 1992.

[5] J. Cohen, "Statistical power analysis for the behavioral science," 1988.

[6] Y. Dang, D. Zhang, S. Ge, C. Chu, Y. Qiu, and T. Xie, "Xiao: Tuning code clones at hands of engineers in practice," in *Proceedings of the 28th Annual Computer Security Applications Conference*, 2012, pp. 369–378.

[7] H. P. Friedman and J. Rubin, "On some invariant criteria for grouping data," *Journal of the American Statistical Association*, vol. 62, no. 320, pp. 1159–1178, 1967.

[8] N. Göde and R. Koschke, "Incremental clone detection," in *Proceedings of the 13th European Conference on Software Maintenance and Reengineering (CSMR)*, 2009, pp. 219–228.

[9] S. Grant and J. Cordy, "Vector space analysis of software clones," in *Proceedings of International Conference on Program Comprehension (ICPC)*, 2009, pp. 233–237.

[10] J. A. Hartigan and M. A. Wong, "Algorithm AS 136: A K-Means Clustering Algorithm," *Applied Statistics*, vol. 28, no. 1, pp. 100–108, 1979.

[11] Y. Higo, K. Tanaka, and S. Kusumoto, "Toward identifying inter-project clone sets for building useful libraries," in *Proceedings of the 4th International Workshop on Software Clones*, 2010, pp. 87–88.

[12] B. Hummel, E. Juergens, L. Heinemann, and M. Conradt, "Index-based code clone detection: incremental, distributed, scalable," in *Proceedings of International Conference on Software Maintenance (ICSM)*, 2010, pp. 1–9.

[13] J. W. Hunt and T. G. Szymanski, "A fast algorithm for computing longest common subsequences," *Commun. ACM*, vol. 20, no. 5, pp. 350–353, May 1977.

[14] 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 *Proceedings of the 19th Working Conference on Reverse Engineering (WCRE)*, Oct 2012, pp. 387–391.

[15] J. C. J. Romano, J.D. Kromrey and J. Skowronek, "Appropriate statistics for ordinal level data: Should we really be using t-test and cohen's d for evaluating group differences on the nsse and other surveys?" in *AIR Forum*, 2006, pp. 1–33.

[16] 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*, ser. ICSE '07, 2007, pp. 96–105.

[17] P. L. Kai Chen and Y. Zhang, "Achieving accuracy and scalability simultaneously in detecting application clones on android markets," in *IEEE International Conference on Software Engineering*, 2014.

[18] I. Keivanloo, J. Rilling, and P. Charland, "Internet-scale real-time code clone search via multi-level indexing," in *Proceedings of the 18th Working Conference on Reverse Engineering (WCRE)*, 2011, pp. 23–27.

[19] I. Keivanloo, J. Rilling, and Y. Zou, "Spotting working code examples," in *Proceedings of the 36th International Conference on Software Engineering (ICSE)*, 2014, pp. 664–675.

[20] I. Keivanloo, C. K. Roy, and R. Juergen, "Sebyte: Scalable clone and similarity search for bytecode," *Science of Computer Programming*, vol. 95, Part 4, no. 0, pp. 426–444, 2014.

[21] R. Koschke, "Large-scale inter-system clone detection using suffix trees," in *Proceedings of the 16th European Conference on Software Maintenance and Reengineering (CSMR)*, March 2012, pp. 309–318.

[22] R. Koschke, R. Falke, and P. Frenzel, "Clone detection using abstract syntax suffix trees," in *Proceedings of Working Conference on Reverse Engineering*, 2006.

[23] S. Livieri, Y. Higo, M. Matushita, and K. Inoue, "Very-large scale code clone analysis and visualization of open source programs using distributed ccfinder: D-ccfinder," in *Proceedings of the 29th International Conference on Software Engineering (ICSE)*, 2007, pp. 106 –115.

[24] G. Milligan and M. Cooper, "An examination of procedures for determining the number of clusters in a data set," *Psychometrika*, vol. 50, no. 2, pp. 159–179, 1985.

[25] J. Ossher, H. Sajnani, and C. Lopes, "File cloning in open source java projects: The good, the bad, and the ugly," in *Proceedings of the 27th IEEE International Conference on Software Maintenance (ICSM)*, ser. ICSM '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 283–292.

[26] K. J. Ottenstein, "An algorithmic approach to the detection and prevention of plagiarism," *SIGCSE Bull.*, vol. 8, no. 4, pp. 30–41, Dec. 1976.

[27] A. Panichella, B. Dit, R. Oliveto, M. Di Penta, D. Poshyvanyk, and A. De Lucia, "How to effectively use topic models for software engineering tasks? an approach based on genetic algorithms," in *Proceedings of the 2013 International Conference on Software Engineering*, 2013.

[28] J.-w. Park, M.-W. Lee, J.-W. Roh, S.-w. Hwang, and S. Kim, "Surfacing code in the dark: an instant clone search approach," *Knowledge and Information Systems*, pp. 1–33, 2013.

[29] C. Roy and J. 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*, ser. ICPC '08, 2008, pp. 172–181.

[30] Y. Sasaki, T. Yamamoto, Y. Hayase, and K. Inoue, "Finding file clones in freebsd ports collection," in *Proceedings of the 7th IEEE Working Conference on Mining Software Repositories (MSR)*, May 2010, pp. 102–105.

[31] N. Schwarz, M. Lungu, and R. Robbes, "On how often code is cloned across repositories," in *Proceedings of the 34th International Conference on Software Engineering*, 2012, pp. 1289–1292.

[32] D. J. Sheskin, *Handbook of Parametric and Nonparametric Statistical Procedures, Fourth Edition*. Chapman & Hall/CRC, Jan. 2007.

[33] J. Svajlenko, J. F. Islam, I. Keivanloo, C. K. R. Roy, and M. M. Mia, "Towards a big data curated benchmark of inter-project code clones," in *Proceedings of the 30th International Conference on Software Maintenance*, ser. ICSM 2014, 2014.

[34] M. Uddin, C. Roy, K. Schneider, and A. Hindle, "On the effectiveness of simhash for detecting near-miss clones in large scale software systems," in *roceedings of the 18th Working Conference on Reverse Engineering (WCRE)*, 2011, pp. 13–22.

[35] T. Wang, M. Harman, Y. Jia, and J. Krinke, "Searching for better configurations: A rigorous approach to clone evaluation," in *Proceedings of the 9th Joint Meeting on Foundations of Software Engineering*, 2013.

[36] R. K. Yin, *Case Study Research: Design and Methods - Third Edition*, 3rd ed. SAGE Publications, 2002.

[37] F. Zhang, A. Mockus, I. Keivanloo, and Y. Zou, "Towards building a universal defect prediction model," in *Proceedings of the 11th Working Conference on Mining Software Repositories*, ser. MSR '14, 2014, pp. 41–50.