Efficiently Measuring an Accurate and Generalized Clone Detection Precision using Clone Clustering

Jeffrey Svajlenko Chanchal K. Roy
Department of Computer Science, University of Saskatchewan, Saskatoon, Canada
{jeff.svajlenko,chanchal.roy}@usask.ca

Abstract—**An important measure of clone detection performance is precision. However, there has been a marked lack of research into methods of efficiently and accurately measuring the precision of a clone detection tool. Instead, tool authors simply validate a small random sample of the clones their tools detected in a subject software system. Since there could be many thousands of clones reported by the tool, such a small random sample cannot guarantee an accurate and generalized measure of the tool's precision for all the varieties of clones that can occur in any arbitrary software system. In this paper, we propose a machine-learning based approach that can cluster similar clones together, and which can be used to maximize the variety of clones examined when measuring precision, while significantly reducing the biases a specific subject system has on the generality of the precision measured. Our technique reduces the efforts in measuring precision, while doubling the variety of clones validated and reducing biases that harm the generality of the measure by up to an order of magnitude. Our case study with the NiCad clone detector and the Java class library shows that our approach is effective in efficiently measuring an accurate and generalized precision of a subject clone detection tool.**

I. Introduction

Clones are pairs of code fragments that are similar. Developers create clones when they reuse code using copy and paste, although clones may arise for a variety of other reasons [1]. Clone detection tools locate clones within or between software systems. Developers need to detect and manage their clones in order to maintain software quality, detect and prevent bugs, reduce development risks, and so on [1]. It is therefore important that the developers have high-quality clone detection tools, which requires knowledge of their detection performance.

Clone detection tools are evaluated in terms of recall and precision. Recall is the ratio of the clones within a software system a tool is able to detect, and precision is the ratio of the detected clones that are true positives, not false positives. While high-quality benchmarks have been proposed for measuring recall [2]–[4], accurately measuring precision remains difficult. In general, there has been a marked lack of research into methodologies for measuring a generalized precision both accurately and efficiently.

Precision is typically measured by validating a random sample of the clones the tool detects within a software system. For example, tool authors have checked on the order of 100 clones detected by their tool [5], [6]. However, this leads to a precision that is not generalizable and therefore not accurate. While a tool often detects a diverse variety of clones within a software system, the detection report is often dominated by a few large groups of similar clones. These groups are distinct varieties of clone pairs that are common in the subject system, and are similar in terms of clone validation, but are not necessarily clones of each other (clone classes). A random sampling will mostly select from these similar clones, and a significant variety of clones are missed. This biases the measurement of precision to the varieties of clones that are most common in this subject system, but which may be rare in another software system. The result is a precision measurement that is not generalizable to another arbitrary software system, and is therefore not an accurate or useful measure for the users of the tool. While the variety of clones examined could be increased by sampling clones from a variety of software systems, this is difficult because clone validation is a very time-consuming process. Previous studies have had difficulty measuring precision for more than two [7] to eight [8] subject systems. The reliability of judges is also a major concern [9]–[11], so one cannot simply hire a large number of non-experts (e.g., undergraduate students) to scale the task. Random sampling is simply not an efficient way to select a variety of clones for measuring precision.

We propose a novel machine-learning-based approach for selecting a better sample of clones for measuring an accurate and generalized precision. This approach involves unsupervised clustering of the similar clone pairs detected in a software system (or collection of software systems) together. The goal is to cluster together the clone pairs that are similar in terms of the syntax and/or semantics that causes them to be validated as true or false positives, and which should be considered a distinct variety of clones when measuring precision. This is a finer-granularity concept than clone types. Precision can then be measured by validating a randomly chosen exemplar clone from each cluster. This efficiently maximizes the variety of the clone pairs examined when measuring precision. It also minimizes the bias caused by the particular distribution of the varieties of clones in the specific subject system, allowing a more generalized result to be obtained.

We evaluate our approach by clustering the clones detected by NiCad in the Java class library using K-means and measure precision. We compare this against the random sampling approach. Our approach doubles the varieties of clones examined within a sample size of 100 clones, while reducing biases in the measurement by up to an order of magnitude.

DOI reference number: 10.18293/SEKE2016-150

II. Definitions

Code Fragment: A continuous segment of source code, specified by the triple (l, s, e), including the source file, l, the line the fragment starts on, s, and the line it ends on, e.
Clone Pair: A pair, (f_1, f_2), of similar code fragments.
Clone Class: A set of similar code fragments. Specified by the tuple $(f_1, f_2, ..., f_n)$. Each pair of distinct code fragments is a clone pair: $(f_i, f_j),\ i, j \in 1..n,\ i \neq j$.
Type-1 Clone: Identical code fragments, except for differences in white space, layout and comments [1], [8].
Type-2 Clone: Identical code fragments, except for differences in identifier names and literal values, in addition to Type-1 clone differences [1], [8].
Type-3 Clone: Similar code fragments that differ at the statement level. The fragments have statements added, modified and/or removed with respect to each other, in addition to Type-1 and Type-2 clone differences [1], [8]. There is no agreement on the minimum similarity of a Type-3 clone [1].

III. Clustering Background

Clustering is an unsupervised learning technique for grouping similar objects. The goal is to group the objects such that an object is more similar to those within its own group, called a cluster, than those in other clusters. Given a dataset of n objects (data points), where each object is represented by a d-dimensional vector of measured features; the clustering problem is to label each data point with a value $[1, k]$, for a target number of clusters k. The data labels constitute a *clustering solution* to the data. Most algorithms require the data to be formatted in a n by d matrix of numerical values and for a number of clusters, k, to be specified.

A. Silhouette Metric

The silhouette metric [13] measures the quality of a clustering solution by how well each data point is clustered. The silhouette of data point i is shown in Eq. 1, where $a(i)$ is the average distance between data point i and the other points in its own cluster, and $b(i)$ is the lowest average distance between data point i and the data points in the other clusters. $a(i)$ measures how dissimilar the data point is to other members of its own cluster, while $b(i)$ measures how dissimilar the data point is to its most similar neighboring cluster. The silhouette of a data point ranges from -1.0 to 1.0. In this study we measure similarity and dissimilarity using the cosine similarity metric (Eq. 3).

$$s(i) = \frac{b(i) - a(i)}{max\{a(i), b(i)\}} \tag{1}$$

A silhouette closer to 1.0 indicates that data point i is much more similar to the data points in its own cluster than those in its neighboring clusters ($a(i) << b(i)$), and is appropriately clustered. A silhouette closer to -1.0 indicates the data point is much more similar to data points in a neighboring cluster than those within its own cluster ($a(i) >> b(i)$), and would be more appropriately placed in the neighboring cluster. A value closer to 0.0 indicates that the data point lies on the border of two clusters ($a(i) \sim= b(i)$). The silhouette of a clustering solution is measured as the average silhouette of its data points. This measures how well the data has been clustered, with a value closer to 1.0 being preferable.

The silhouette has a flaw that we must control for. If a cluster is created per unique data point, the silhouette trivially returns 1.0, a "perfect" clustering. Although such a clustering is unlikely to be useful. We want to cluster similar clones, not only identical clones. To control for this we also measure the *percentage of singleton clusters*. This is the ratio of the clusters that contain only a single unique data point (although they may contain multiple duplicate data points). We use this metric to determine if an increase in silhouette by adding additional clusters is due to a more natural clustering or due to an increase in singleton clusters.

B. Choosing a Number of Clusters

Often clustering is performed on data where the number of classifications in the data is unknown, as is the case with our clone data. A technique must be used to choose the number of clusters. In this paper we use the 'elbow method' [14] to estimate the natural number of clusters in the data. This involves plotting the quality (silhouette) of the clustering solutions as a function of the number of clusters, k, then looking for an 'elbow' in the plot where the gain in cluster silhouette by adding additional clusters suddenly and significantly drops. This estimates the natural number of clusters as the point where adding additional clusters no longer significantly improves the quality of the clustering solution.

C. K-Means Clustering with K-Means++ Initialization

K-means [15], [16] is an iterative clustering algorithm that aims to partition the data in a way that minimizes a loss function: the within-cluster sums of squares as shown in Eq. 2. Where k is the number of clusters, C_i is the set of data points in cluster i, $\vec{\mu}_i$ is the center of cluster i, and $d(\vec{x}, \vec{\mu}_i)$ is the distance between data point $\vec{x}$ in cluster C_i and the cluster's center $\vec{\mu}_i$. In document clustering, where documents are represented as weighted term-frequency vectors, the cosine distance, Eq. 3, is the preferred distance metric [17]. The algorithm is guaranteed to converge to a local optimum, although this may not be the global optimum.

$$\sum_{i=1}^{k} \sum_{x \epsilon C_i} \|d(\vec{x}, \vec{\mu}_i)\|^2 \tag{2}$$

$$d(\vec{a}, \vec{b}) = 1 - cos(\theta) = 1 - \frac{\vec{a} \cdot \vec{b}}{||\vec{a}||\ ||\vec{b}||} \tag{3}$$

Given an initial set of cluster centers, the K-means algorithm iteratively updates the cluster centers to a local minimum of the loss function using the algorithm below. The clustering solution returned depends on the number of clusters and the initial cluster centers used.

1) Assign each data point to its nearest cluster center using the chosen distance measure.
2) Update the cluster centers as the mean of the points assigned to them.
3) Repeat steps 1-2 until the sums of squares converges or a maximum number of iterations has been reached.

We initialize the cluster centers using the K-means++ algorithm [18]. It aims to avoid poor clusterings by choosing random but evenly distributed cluster centers amongst the data. It guarantees a clustering solution that is at least $O(\log k)$ competitive with the optimal solution, and generally improves the speed and accuracy of K-means [18]. The algorithm chooses clustering centers using the following procedure, where D is the dataset:

1) Choose one $x \in D$ with uniform probability to be the first initial cluster center.
2) For each data point $x \in D$ the distance, $d(x)$, between x and its nearest previously chosen initial cluster center is measured.
3) Choose the next initial cluster center from $x \in D$ with a probability of choosing x of $\frac{d(x)^2}{\sum_{y \in D} d(y)^2}$.
4) Repeat steps 2-3 until k initial cluster centers have been chosen.

D. *Principal Component Analysis (PCA)*

Principle component analysis [19] is an orthogonal linear transformation of the data onto a new basis of linearly uncorrelated axes called principal components. These axes are chosen in decreasing order of the greatest data variance they explain. The dimensionality of the transformed data can be reduced by dropping the principal components that explain the least variance in the original data. Since the principal components are ordered by variance explained, only the Top-T principal components are kept for some target preservation of total variance explained.

IV. Clone Variety

We define a **clone variety** as a collection of clones that are similar in terms of their clone validation for measuring precision. Two clone pairs are of the same clone variety if they share the same syntax, code-patterns and/or semantics that determine their validation as a true or false positive clone. This is different from clone type, which is a classification of clones based on the tool features (e.g. normalizations) needed to detect them (recall-focused). Clones of the same clone type can be very different, and be unrelated in how they are validated to measure precision.

As an example, clone detectors often report simple constructors as clones, that take a number of arguments and initialize member fields with their values. The clone detector reports similar pairs of these simple constructors as clones. These clone pairs are not necessarily clones of each other. They may vary by length, contents (e.g., names and number of parameters), and may be of different clone types. However, in all cases, validation depends on the decision of if two simple constructors form a true clone, so we consider these clones to be of the same variety of clone. Some other clone varieties observed in this study include: clones of methods that register action listeners, of auto-generated equals() methods, of methods implementing buffer slicing, and so on. While these clones can be found in many software systems, their frequency of occurrence depends on the subject system in question.

When measuring precision, we ideally only want to validate a single exemplar from each clone variety, as additional exemplars do not tell us anything new about the clone detector's precision. To measure a generalized precision, we want to give each variety of clone equal weighting, as different clone varieties may be more common or more rare in different subject systems. A generalized precision should be measured for a flat distribution of the different clone varieties found.

We do not attempt to build a taxonomy of all varieties of clones from the perspective of clone validation. Across the entire software development community, there is likely an unlimited number of clone varieties, and these varieties may overlap. Our interest is simply to judge if a cluster contains a single or multiple varieties of clones, and to measure the total number of clone varieties in a sample of detected clones.

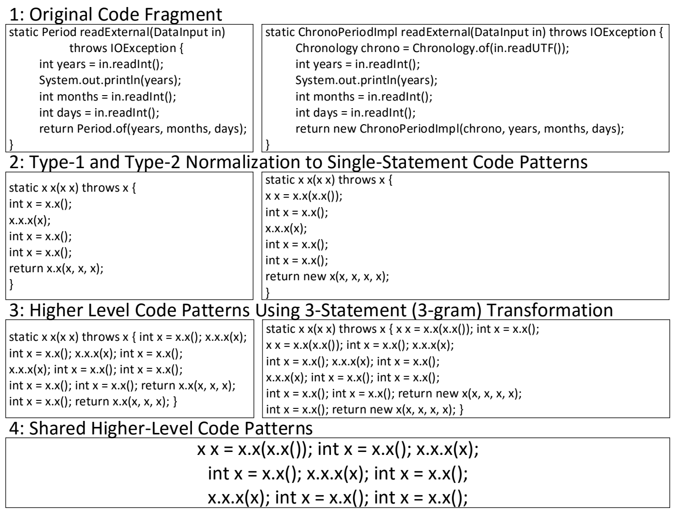
V. Measuring Precision with Clone Clusters

Here we discuss how precision can be measured using a clone clustering solution. The subject clone detector is executed for one or more subject software systems and the detected clone pairs are clustered. The goal is to produce a clustering solution where the clone pairs of the same clone variety are placed in the same cluster, and the clone pairs of different varieties are placed in different clusters. Precision is then measured by randomly selecting an exemplar clone pair from each cluster and manually validating them as true or false clones. The ratio of the exemplar clones judged as true clones by a clone expert is the precision of the tool.

This procedure maximizes the variety of the clones considered when measuring precision. It minimizes biases in the measured precision caused by the particular distribution of clone varieties in the particular subject systems under test, therefore measuring a more generalizable precision with respect to any arbitrary software system. It also minimizes the number of clone pairs that must be validated to achieve a desired variety of clones examined when measuring precision.

We cluster the clone pairs as a whole unit, not their individual code fragments. We cluster clone pairs rather than clone classes because clone detectors might mix true and false positives, and different varieties of clone pairs, within the same clone class. Clone classes can be unwieldy to validate, and there is no standard way of validating a whole clone class. Additionally, not all tools support clone class output, while all clone detection tools report clone pairs. We do not cluster clones of different clone types separately as it is not typical to consider them separately when measuring precision [1].

It is not realistic to expect a perfect clustering. The clustering solution might split clone varieties across multiple clusters, which can cause the clone varieties to have an uneven influence on the precision measured. The clustering solution may cluster multiple clone varieties into the same cluster, which may cause some clone varieties to be missed when choosing the exemplar clone pairs. Despite this, the goal is to achieve a better result than measuring precision by a pure random sampling of the detected clones. In our experiment, we find our approach achieves twice the clone variety in the same sample size,



1: Original Code Fragment

```
static Period readExternal(DataInput in)
            throws IOException {
      int years = in.readInt();
      System.out.println(years);
      int months = in.readInt();
      int days = in.readInt();
      return Period.of(years, months, days);
}
```

```
static ChronoPeriodImpl readExternal(DataInput in) throws IOException {
      Chronology chrono = Chronology.of(in.readUTF());
      int years = in.readInt();
      System.out.println(years);
      int months = in.readInt();
      int days = in.readInt();
      return new ChronoPeriodImpl(chrono, years, months, days);
}
```

2: Type-1 and Type-2 Normalization to Single-Statement Code Patterns

```
static x x(x x) throws x {
int x = x.x();
x.x.x(x);
int x = x.x();
int x = x.x();
return x.x(x, x, x);
}
```

```
static x x(x x) throws x {
x x = x.x(x.x());
int x = x.x();
x.x.x(x);
int x = x.x();
int x = x.x();
return new x(x, x, x, x);
}
```

3: Higher Level Code Patterns Using 3-Statement (3-gram) Transformation

```
static x x(x x) throws x { int x = x.x(); x.x.x(x);
int x = x.x(); x.x.x(x); int x = x.x();
x.x.x(x); int x = x.x(); int x = x.x();
int x = x.x(); int x = x.x(); return x.x(x, x, x);
int x = x.x(); return x.x(x, x, x); }
```

```
static x x(x x) throws x { x x = x.x(x.x()); int x = x.x();
x x = x.x(x.x()); int x = x.x(); x.x.x(x);
int x = x.x(); x.x.x(x); int x = x.x();
x.x.x(x); int x = x.x(); int x = x.x();
int x = x.x(); int x = x.x(); return new x(x, x, x, x);
int x = x.x(); return new x(x, x, x, x); }
```

4: Shared Higher-Level Code Patterns

```
x x = x.x(x.x()); int x = x.x(); x.x.x(x);
int x = x.x(); x.x.x(x); int x = x.x();
x.x.x(x); int x = x.x(); int x = x.x();
```

Fig. 1: Clone-Pair to Single-Document Conversion

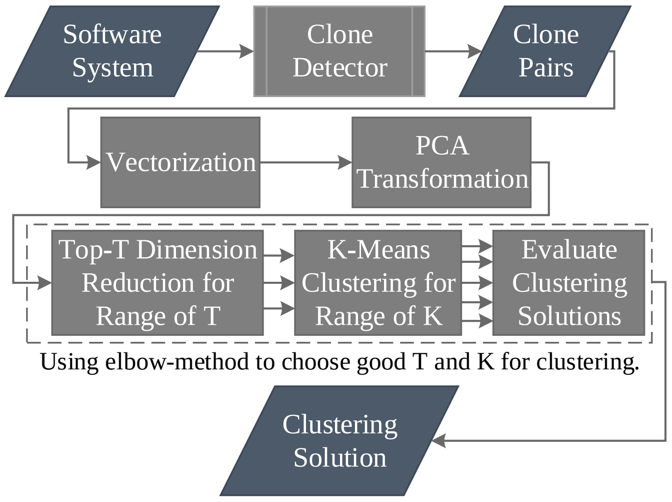

Software System
Clone Detector
Clone Pairs
Vectorization
PCA Transformation
Top-T Dimension Reduction for Range of T
K-Means Clustering for Range of K
Evaluate Clustering Solutions
Using elbow-method to choose good T and K for clustering.
Clustering Solution

Fig. 2: Overview of Experimental Procedure

while reducing biases that affect the generality of the measured precision by up to an order of magnitude.

VI. Clone Vectorization

Clustering algorithms require the data to be represented by a n by d matrix of numerical values, where n is the number of data points (clone pairs) and d is the number of features per data point. To convert the clones into this format we use standard document vectorization [17]. This treats a document as a bag-of-terms, and represents it as a vector of term-frequencies in term-space. The vector is weighted by the inverse-document-frequencies of the terms, under the assumption that frequent terms have less discriminating power. A document has the vector form shown in Eq. 4, where tf_i (term-frequency) is the frequency of term i in the document, df_i (document-frequency) is the number of documents term i appears in, n is the total number of documents, and d unique terms occur across all of the documents.

$$document = \{tf_1 \log \frac{n}{df_1}, tf_2 \log \frac{n}{df_2}, ..., tf_d \log \frac{n}{df_d}\} \quad (4)$$

Clones are pairs of similar code fragments (documents), so to apply document vectorization we need to covert the clone pairs into a single-document representation and choose the granularity of a term. From the definitions, the most important aspect of a clone is the code shared by its code fragments. The clone types indicate that the Type-1 and Type-2 differences in the clones should be ignored, so we normalize these differences in the code fragments, turning them into sequences of statement-level code patterns. However, it can be trivial for clones to share normalized code statements. Instead, we capture higher-level code patterns as sequences of multiple statements by applying an n-gram transformation over the normalized code statement patterns. Treating each code fragment as a bag of n-statement normalized code patterns, we extract the shared higher-level code patterns by computing the intersection of these bags, allowing duplicates. The single-document representation is then this bag of shared higher-level code patterns. An example of this is shown in Figure 1 for n=3. We drop the terms that appear in only a single clone pair as it discourages the clustering of similar clones, and singleton clusters are not desirable in our case. This single-document version is then vectorized as shown in Equation 4, with the n-statement code patterns as terms.

The goal is to cluster clone pairs of the same clone variety (Section IV), which we defined as the high-level shared syntax and semantics that determine the validation of the clone pair. We extract a clone pair's significant high-level cloned code patterns that define its clone variety. Then clustering is used to group it with clone pairs of the same clone variety, even if they are not clones of each other. The vectorization process is designed to only consider the significant cloned features of a clone pair, not the fine-grained cloned and non-cloned features that define variance within a clone variety.

VII. Clone Clustering Procedure

In this section we describe the clone clustering procedure, which is summarized in Figure 2. For clustering, we use the K-means algorithm because of its availability in most data analysis frameworks. Our intention is for this methodology to be adopted by the cloning community, so it is important to use a algorithm which is accessible. In our experiment, we find K-means provides a good result for our use-case.

First, the clone detection tool is executed for the subject software system(s) and the detected clone pairs are collected. The clone pairs are then vectorized into d-dimensional vectors of real-numbers, as described in Section VI. The vectorized clones are of a very high dimensionality. Clustering algorithms are known to perform poorly in high dimensions [20]. In high dimensions, distance metrics begin to return similar values for any arbitrary pair of data vectors due to a high degree of orthogonality, which can cause poor clustering results with K-means. We perform dimensionality reduction using principal component analysis (PCA, Section III-D). First the clone vectors are rotated onto their principal components in order of decreasing variance explained, and then only the Top-T

principal components (dimensions) are kept for some target total variance explained. K-means (Section III-C) is executed on the transformed/reduced clone vectors to produce the clustering solution for a target number of clusters K. K-means is initialized using the K-means++ algorithm and performed using the cosine distance metric, which is the recommended metric for document-type data [17].

The final steps of the procedure require choosing T, the number of principal components to keep in dimensionality reduction, and K, the number of clusters. We have developed a procedure for choosing good T and K values based on the elbow-method (Section III-B) with the *silhouette* clustering quality metric (Section III-A).

First, multiple versions of the clone data vectors are produced for different Top-T dimensional reductions. For example, the Top-T principal components to preserve 50-90% (in increments of 10%) of the total data variance explained. For each Top-T, K-means is executed across a large range of K values, and for each clustering solution the silhouette is measured. The range of K should be chosen experimentally in order to observe the elbow in the plot of the silhouette versus the number of clusters. For each T considered, a plot of silhouette versus K is produced, and good values of T and K can be chosen from these plots considering the elbow method.

The value of T is chosen as the Top-T reduction that produces a stable plot of silhouette versus K with a well-defined elbow. We have observed that when too many principal components are kept, weak or multiple elbows are observed. When too few principal components are kept, the plot becomes unstable and no clear elbows are observed. So T is chosen as the reduction that produces the best plot with a clear elbow, and K is chosen as the number of clusters at the elbow. The clustering solution produced with the empirically chosen T and K parameters is used as the final clustering solution for the clones. While this procedure does not guarantee the best K and T values, and the silhouette metric does not understand the semantic concept of clone varieties, we find that it results in a good clustering solution for our use-case. It is not possible to manually inspect every clustering solution to pick the best K and T, and this procedure leads to a more natural clustering than choosing K and T arbitrarily.

VIII. Experiment

We test our approach by clustering the clones NiCad [21] detects in the top-level 'java' package of the Java 8 class library. We use the Java class library as it is a common target in clone studies [3], [8], and because it contains a wide variety of functionalities and therefore a wide variety of clones. We use the NiCad clone detector as it is a popular state of the art clone detector with strong recall for all three of the primary clone types [3], [22]. This ensures we are detecting a wide variety of clones for evaluating our approach.

We executed NiCad for the detection of function-granularity clones of 6 lines or greater with no more than 30% dissimilarity after blind identifier normalization and literal abstraction. This is a generous configuration which ensures we detect a wide variety clones. In total, 14,076 clones pairs were detected. We vectorized the clones using a 3-statement (3-gram) representation. This is a compromise between capturing high-level code patterns, and being mindful that Type-3 clones contain statement-level gaps that may split sequences of shared code statements. The vectors are over 2711 term dimensions. This is a reduction from 7719 term dimensions after removing the singleton terms.

We use MatLab to perform principal component analysis on the clone data. This returns the vectors rotated onto a linearly uncorrelated basis of their principal components in decreasing order of the data variance explained by the principal components. Essentially 100% of the variance is explained by the first 1000 of the 2711 principal components, while 90% of the variance is explained by just the first 250 principal components.

We used the procedure described in Section VII to experimentally choose values of T and K. We explored values of T for preserving 50-90% of the total variance in the data. This corresponds to convenient T values of 15 (56%), 25 (63%), 50 (71%), 100 (80%) and 250 (90%). For each of these Top-T dimension reductions of the PCA clone vectors, we performed K-means clustering for a range of K from 20 to 1300. We executed K-means using MatLab with the cosine distance metric and initialized by the K-means++ algorithm, as previously described. We plot cluster silhouette versus K for each Top-T reduction in Figure 3.

The reduction to the first T=100 principal components (80% of total variance explained) appears to be the ideal reduction. It has a single well-defined elbow (natural clustering point), while the other reductions either have unclear or multiple elbows, or are otherwise unstable. It achieves a higher silhouette than keeping more dimensions (250), while having comparable silhouette to the further reductions (15-50).

We plot silhouette and percentage of singleton clusters versus K for K-means solutions with T=100 in Figure 4. The elbow method on the silhouette indicates that K=100 clusters is the natural number of clusters for this data. This is supported by the percentage of singleton clusters. Silhouette improves sharply up to 100 clusters, after which the gain in silhouette by adding additional clusters suddenly declines. As the silhouette slowly increases to 1.0 after 100 clusters, we see the percentage of singleton clusters increases linearly at a significant slope. The (weak) increase in silhouette after 100 clusters is most dominantly due to the increase of singleton clusters, which are not desirable. In contrast, before 100 clusters, the percentage of singleton clusters oscillates without any definite trend. The increase in silhouette as the number of clusters is increased to 100 is most dependent upon approaching a natural clustering point, not the number of singleton clusters.

Therefore our best K-means solution is achieved at T=100 principal components and K=100 clusters. To get our final clustering solution, we re-execute the K-means algorithm with 10 repetitions, meaning K-means is executed 10 times with different K-means++ initializations, and the best clustering,

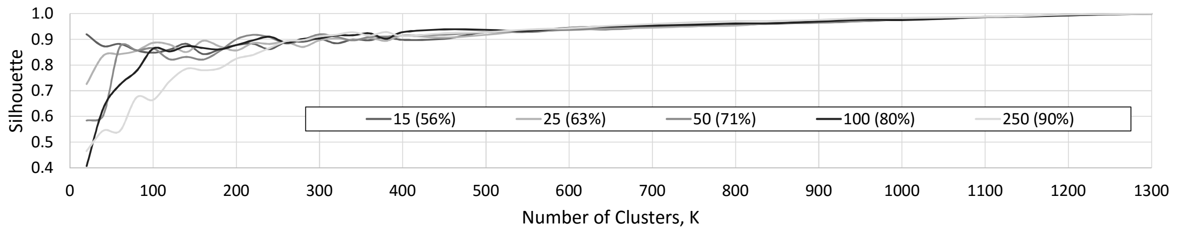


Fig. 3: Silhouette of K-Means Solutions Across k for Different Top-T Dimensionality Reductions

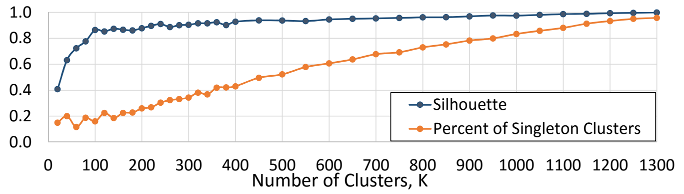


Fig. 4: Evaluation of K-means Clusterings for T=100

with the lowest total within-cluster sum of squares, is returned. This clustering solution has a silhouette of 0.86 with 16% singleton clusters.

IX. Manual Inspection of the Clusters

We now manually inspect the clustering solution to see how well it isolates distinct clone varieties. We define a clone variety as a group of clones that are similar by the code patterns, syntax and/or semantics that determine their validation as true or false clones (Section IV). The goal was to produce a clustering of clones pairs where clones of the same clone variety are placed in the same cluster, and clones of different clone varieties are placed in different clusters. Although a perfect clustering was not expected, nor realistic to obtain. During this inspection we judged the clusters as strong or weak. A "strong" cluster is one that contains only a single variety of clones, while a "weak" cluster contains multiple varieties of clones. We also investigated if clone varieties were contained within a single strong cluster, or split across multiple. This manual inspection was done to judge the quality of the clustering, and is not a required component of the procedure for measuring precision (Section V).

When judging the clusters as strong or weak, we considered the syntax, semantics and code patterns of the clone pairs. In particular, how these elements affect the decision to validate the clones as true or false positives from the perspective of a variety of clones. A particular clone variety may span multiple clone types, and clones of the same variety may not be clones of each other. A few examples of the strong clusters we found include: clones of constructors that take a number of parameters and initialize fields with these values, clones of functions implementing buffer slicing on arrays of different primitive data types, and clones of short file-system operations wrapped in a common Java security manager code-pattern. In a few cases a large cluster of a single variety of clones contained a single or a small number of outliers. We judged these as strong clusters if the number of outliers was very small compared to the cluster size.

We judged 76 of the clusters as strong, meaning they contain only a single variety of clone. However, we noticed that some of the strong clusters should ideally be merged. Specifically, multiple clusters contained clones of the same clone variety. This means that some of the exemplar clones chosen from the strong clusters will be of the same clone variety. The 76 strong clusters yield 56 distinct clone varieties when an exemplar clone is chosen from each of the strong clusters. While the splitting of a clone variety into multiple strong clusters is not a problem in terms of missing a variety of clones when measuring precision in this way, it does needlessly increase the number of exemplars to be examined, and adds a small bias in the overall precision measurement to these clone varieties. In this case it is minor, affecting only 11 of the 56 (20%) clone varieties found in the strong clusters.

We judged 24 of the clusters as weak, meaning they contain multiple varieties of clones. This does not mean the clones within the cluster do not share features, nor that they are all completely disparate. They contain clones that are sufficiently different to be considered different varieties of clones from the perspective of clone validation. When we investigated the weak clusters, we found that they could be converted into strong clusters if appropriately split. In most cases, a weak cluster was the merging of only 2-5 varieties of clones, while others had a more significant number of these hidden strong clusters. Some of these hidden strong clusters are of varieties of clones not previously seen, while some of them would ideally be merged with one of the existing strong clusters. Since some of the weak clusters contain clones not seen in the strong clusters, exemplars chosen from them still contribute to the variety of clones examined for measuring precision.

We randomly selected an exemplar from each of the weak clusters and found that 20 of these 24 clones were distinct from each other and not seen in the strong clusters. This suggests that some clone variety is missed when we only select a single exemplar from the weak clusters. This could be alleviated by selecting multiple exemplars from these clusters, although this is not necessary to measure a high-quality precision. A better use of additional efforts would be to extend the measure of precision to additional subject systems.

Overall, we judged 76 of the 100 clusters as strong and 24 as weak. In absolute number of clone pairs, 10,505 of the 14,069 clones pairs (75%) are in the strong clusters. The strong clusters perfectly capture the clone variety in the majority of the clone detection report, while the weak clusters still capture partial clone variety for the remaining 25%. Selecting one exemplar per cluster, a total sample size of 100 clones, yields 76 distinct varieties of clones: 56 from the strong clusters and 20 from the weak clusters. This is an efficiency of 76% in terms of clone variety within the inspected clone pairs.

X. Evaluation

We now compare our technique against the traditional pure random sampling in terms of clone variety and the biases that affect the accuracy and generality of the precision measure. As we found in the previous section, a sample of clones by choosing a single exemplar from each cluster in our clustering solution yields 76 distinct varieties of clones within a 100 clone sample (76% efficiency). For comparison, we selected 100 random clone pairs from the entire clone dataset (without clustering) and manually grouped them by clone varieties. A sample of 100 clone pairs by pure-random sampling yields only 37 distinct clone varieties (37% efficiency). This is the best-case we saw over several trials of selecting 100 clones at random. Our cluster-based approach achieves over twice the clone variety for the same clone validation efforts. We continued to randomly sample clone pairs until we reached parity. The random sampling approach required a sample size of 272 clone pairs to reach 76 distinct clone varieties at 28% efficiency. The efficiency in selecting a variety of clones by random sampling becomes less efficient as the sample size grows as it becomes more likely to choose varieties already seen. The random sampling approach requires almost three times the manual efforts to capture the same variety of clones. For this same effort, we could execute our cluster-based approach for additional subject systems and further increase the variety of clones considered.

While, with significant additional efforts, the traditional approach can match our approach in terms of total variety of clones considered, it cannot guarantee an accurate and generalized precision due to biases in the clone sample. We compare the distribution of the 76 clone varieties found by our cluster-based approach (100 exemplar clone pairs) and the traditional random sampling (272 randomly sampled clone pairs) in Figure 5. This plot shows the number of times each variety of clones appears in the clone samples produced by the two approaches. The clone varieties are ordered by increasing frequency, but the clone varieties do not necessary correspond between the two techniques. For each approach, we highlight the first variety with a frequency greater than one.

For an accurate and generalized precision, each clone variety ideally has equal weighting in the measurement of precision. The clones considered by both techniques exhibit some bias due to some clone varieties appearing multiple times in the sample, causing these clone varieties to have a stronger weighting in the precision measurement. Our cluster-based technique has 13 clone varieties appearing multiple times in the sample, with most occurring twice, and a worst case of 5 exemplar clone pairs being of the same clone variety. The random sampling technique has 25 clone varieties appearing multiple times in its sample, with a range of re-occurrence of 2 to 64. With random sampling, five of the clone varieties have an order of magnitude higher impact on the precision measurement than the others, with a range of 12x to 64x in the worst case. These clone varieties significantly bias the precision measurement. In contrast, our cluster-based approach reduces these biases by up to an order of magnitude. Overall, our approach exhibits very little bias.

While random sampling can estimate the precision of a clone detector for a specific subject system, it is not effective in measuring a generalized precision. Our cluster-based technique excels in measuring an accurate and generalized precision. Our technique requires a smaller sample size, therefore less efforts, than random sampling.

XI. Measuring NiCad's Precision

To complete the case study, we also measure the precision of NiCad using our cluster-based procedure (Section V). We randomly selected one exemplar clone pair per cluster (100 clones) for manual validation. Since there is no clear and universally accepted definition of what constitutes a true clone, precision must be measured with respect to some context or use-case. In this case, we measured precision from a software maintenance perspective. We judged a clone pair as a true positive if its code fragments were not only syntactically and/or semantically similar, but if it would also be useful from a software maintenance perspective. Specifically, if one of the code fragments needed to be modified to fix a bug or to evolve the code, should the other code fragment also be considered for the same bug fix or code evolution. Otherwise we judged a clone to be a false positive. This is a rather strict criteria for a true positive clone.

Overall, we measured a precision of 74% for NiCad. The identified false positives were code fragments with similar syntax, but were not specifically relevant to software maintenance. In particular, clones of common Java programming and API idioms, as well as clones of auto-generated code. While these clones do share syntax, they are not useful for the identified clone-related software maintenance tasks. NiCad's precision is lower in this case as we purposefully used a generous configuration to ensure we detected a wide variety of both true and false clones to fully evaluate our cluster-based technique. Previous work [23], [24] has measured a precision of 80-90%+ when more conservative or recommended settings are used.

XII. Threats to Validity

A different clustering algorithm might produce a better clustering. We use the K-means algorithm as it is simple and an implementation is available in most data analysis frameworks. This is important as our intention is for this technique to be adopted by the cloning community for measuring precision. We have found that K-means performed well in this case.

We limited our study to one clone detection tool and one subject system as the manual evaluation of the clustering solution and the clone validation efforts are very time intensive. We choose the subject system and tool very carefully to ensure we get representative results to the general case. The Java system library has a wide variety of functionalities from different domains and is therefore an ideal target to test clustering of a wide variety of clones. Of the available modern clone detection tools, NiCad has the best recall for the primary three

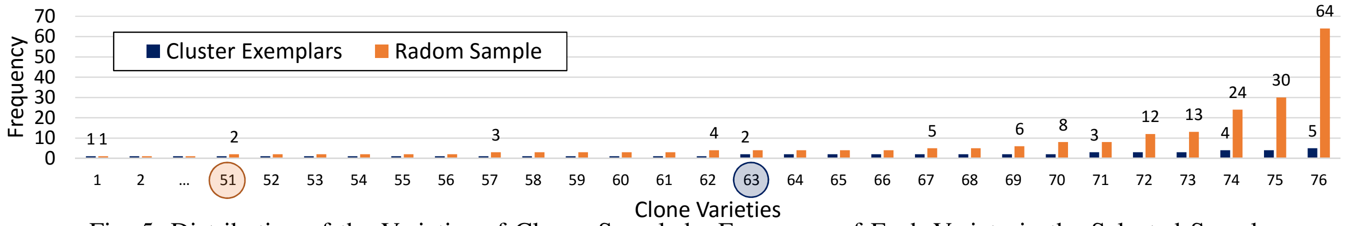


Fig. 5: Distribution of the Varieties of Clones Sampled - Frequency of Each Variety in the Selected Sample

clone types [22]. We configured it to be generous in detecting all types of clones to get the most wide variety of clones. The results should then represent any clone detection tool that targets the first three clone types.

As with any clone study involving manual clone inspection, the analysis, classification and validation of clones can be subjective. We took measures to reduce subjectivity and ensure fairness in the results. When comparing our cluster-based approach against traditional random clone sampling for measuring precision, we needed to classify the varieties of the clones, and measure the total variety of clones seen. We tracked our judgments on the clone varieties and the reasons for these decisions to ensure that the judgments were applied uniformly for the evaluation of both methodologies. Although there is always some subjectivity in manual clone analysis and classification, the comparison of the two approaches was kept fair by ensuring the classifications were applied uniformly.

XIII. Related Work

Various benchmarks [2], [4], [8] have been created and experiments [3], [7], [8], [22] performed for measuring clone detection recall. Precision has been measured by manually validating all clones detected in a very small software system [25] or for a random sample of clones detected in one or more software systems [5]–[8]. Roy and Cordy [23] measured precision in a semi-automatic way specifically for synthetic clones injected into a software system. Yang et al. [26] used supervised learning to classify clones as per a manually labeled dataset. Ours is the first work to use unsupervised clustering to improve the efficiency and accuracy of measuring clone-detection precision.

XIV. Conclusion

We presented a novel approach approach for efficiently measuring an accurate and generalized clone detection precision. This involves clustering the detected clone pairs and validating an exemplar clone pair selected from each cluster. This ensures a wide variety of clones are considered when measuring precision with minimal bias due to the particular distribution of the varieties of clones within the subject system. We compared our approach against traditional random sampling. Our approach samples twice the variety of clones as the traditional approach for the same sample size, while reducing biases that harm the generality of the measured precision by up to an order of magnitude. The clones and clustering solution from this paper are available for interested readers[1].

[1]www.jeff.svajlenko.com/seke16.html

References

[1] C. K. Roy and J. R. Cordy, "A survey on software clone detection research," Queen's University, Tech. Rep. 2007-541, 2007, 115 pp.

[2] 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 *ICSME*, 2014, pp. 476–480.

[3] J. Svajlenko and C. K. Roy, "Evaluating modern clone detection tools," in *ICSME*, 2014, 10 pp.

[4] J. Svajlenko, C. K. Roy, and J. R. Cordy, "A mutation analysis based benchmarking framework for clone detectors," in *IWSC*, 2013, pp. 8–9.

[5] L. Jiang, G. Misherghi, Z. Su, and S. Glondu, "Deckard: Scalable and accurate tree-based detection of code clones," in *ICSE*, 2007, p. 10.

[6] Z. Li, S. Lu, S. Myagmar, and Y. Zhou, "Cp-miner: finding copy-paste and related bugs in large-scale software code," *Software Engineering, IEEE Transactions on*, vol. 32, no. 3, pp. 176–192, March 2006.

[7] R. Falke, P. Frenzel, and R. Koschke, "Empirical evaluation of clone detection using syntax suffix trees," *Empirical Software Engineering*, vol. 13, no. 6, pp. 601–643, 2008.

[8] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, and E. Merlo, "Comparison and evaluation of clone detection tools," *Software Engineering, IEEE Transactions on*, vol. 33, no. 9, pp. 577–591, Sept 2007.

[9] B. Baker, "Finding clones with dup: Analysis of an experiment," *Softw. Eng., IEEE Trans. on*, vol. 33, no. 9, pp. 608–621, 2007.

[10] A. Charpentier, J.-R. Falleri, D. Lo, and L. Réveillère, "An empirical assessment of bellon's clone benchmark," in *EASE*, 2015.

[11] A. Walenstein, N. Jyoti, J. Li, Y. Yang, and A. Lakhotia, "Problems creating task-relevant clone detection reference data," in *WCRE*, 2003.

[12] MathWorks, "Silhouette plot - matlab silhouette," http://www.mathworks.com/help/stats/silhouette.html.

[13] P. J. Rousseeuw, "Silhouettes: A graphical aid to the interpretation and validation of cluster analysis," *Journal of Computational and Applied Mathematics*, vol. 20, pp. 53 – 65, 1987.

[14] DataScienceLab, "Finding the k in k-means," http://datasciencelab.wordpress.com/2013/12/27/finding-the-k-in-k-means-clustering/.

[15] J. B. MacQueen, "Some methods for classification and analysis of multivariate observations," in *Berkeley Symposium on Mathematical Statistics and Probability*, 1967, pp. 281–297.

[16] S. Lloyd, "Least squares quantization in pcm," *IEEE Trans. Inf. Theor.*, vol. 28, no. 2, pp. 129–137, 2006.

[17] M. Steinbach, G. Karypis, and V. Kumar, "A comparison of document clustering techniques," in *KDD Workshop on Text Mining*, 2000.

[18] D. Arthur and S. Vassilvitskii, "K-means++: The advantages of careful seeding," in *SODA*, 2007, pp. 1027–1035.

[19] J. E. Jackson, *A User's Guide to Principal Components*, ser. Wiley Series in Probability and Statistics. Wiley-Interscience, 2004.

[20] A. Rajaraman and J. D. Ullman, "Mining of massive datasets." New York, NY, USA: Cambridge University Press, 2011, ch. 7.

[21] C. K. Roy and J. R. Cordy, "Nicad: Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization," in *ICPC*, 2008, pp. 172–181.

[22] J. Svajlenko and C. Roy, "Evaluating clone detection tools with bigclonebench," in *ICSME*, 2015, pp. 131–140.

[23] C. Roy and J. Cordy, "A mutation/injection-based automatic framework for evaluating code clone detection tools," in *ICSTW*, 2009, pp. 157–166.

[24] H. Sajnani, V. Saini, J. Svajlenko, C. K. Roy, and C. V. Lopes, "Sourcerercc: Scaling code clone detection to big code," in *ICSE'16*, Austin, Texas, 2016.

[25] E. Burd and J. Bailey, "Evaluating clone detection tools for use during preventative maintenance," in *SCAM*, 2002, pp. 36–43.

[26] J. Yang, K. Hotta, Y. Higo, H. Igaki, and S. Kusumoto, "Classification model for code clones based on machine learning," *Empirical Software Engineering*, vol. 20, no. 4, pp. 1095–1125, 2015.