International Journal of Software Engineering and Knowledge Engineering

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

Jeffrey Svajlenko
Department of Computer Science, University of Saskatchewan,
176 Thorvaldson Building, 110 Science Place
Saskatoon, Saskatchewan S7N 5C9,Canada jeff.svajlenko@usask.ca
http://www.jeff.svajlenko.com

Chanchal K. Roy
Department of Computer Science, University of Saskatchewan,
176 Thorvaldson Building, 110 Science Place
Saskatoon, Saskatchewan S7N 5C9,Canada chanchal.roy@usask.ca
http://www.cs.usask.ca/∼croy/

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.

Keywords: clone; clone detection; precision; clone clustering, clone variety

1. Introduction

Code clones are pairs of code fragments that are similar. Developers create clones when they reuse code using copy, paste and modify; 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 developer risks, and so on [1]. Clone detectors are also used to study software systems and mine software repositories, including: mining for new library candidates [2], detecting similar applications [3] and forks [4], detecting license violations within software communities [5],

code search and recommendation [6], and so on. 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. Where recall is the ratio of the clones within a software system a tool is able to detect, and precision is the ratio of the reported clones that are true positives, not false positives. While high-quality benchmarks have been recently proposed for measuring recall [7, 8, 9], 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 can be measured by executing the clone detector for a software system and then manually validating the detected clones. This is repeated for a large number of software systems across a variety of software domains. This is done to expose the tool to a large variety of true and false clones. By variety we do not just mean the different clone types, but the different ways those clone types can be expressed, which determines if they are true or false positives. Precision is measured for each software system as the ratio of the detected clones that are judged to be true positives. The clone detector's overall precision is the average of its precision across all of the subject software systems. The average is taken to normalize for the bias the distribution of the clone varieties in a particular subject system has on the measurement of precision. A particular variety of clone could make up a large proportion of the detected clones in one system, but be rare or never appear in another system. By normalizing across a variety of software systems the measurement is then the expected precision of the clone detector for any arbitrary software system.

The problem is that clone validation is a very time consuming process, and even smaller software systems can contain thousands to tens of thousands of clone pairs [1, 10]. Validating the clones detected in even a single software system can be exhausting, while validating the clones from a variety of systems becomes infeasible due to the time and effort required. This is exasperated by the fact that clone validation is very subjective, and likely the clones should be validated by multiple clone experts to reflect multiple opinions on what constitutes a true positive clone. The reliability of judges is also a major concern [11, 12, 13], so one cannot simply hire a large number of non-experts (e.g., undergraduate students) to scale the task.

Instead, precision is typically approximated 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 [14, 15]. 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 (i.e., 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 or non-existent in another software system. The result is a precision measurement that is not generalizable to another arbitrary software system, and therefore not an accurate or useful measure for the users of the tool. While variety of clones examined could be increased by sampling clones from a variety of software systems, this returns to the problem of too many clones to validate. Previous studies have had difficulty measuring precision for more than two [16] to eight [17] subject systems. Random sampling is simply not an efficient way to select a variety of clones for measuring precision.

In this paper, we propose a novel machine-learning approach for efficiently measuring an accurate and generalized precision. This approach involves clustering the similar clone pairs (and false positives) detected in a software system (or collection of software systems) together. The goal is to cluster the clone pairs that are similar in terms of the semantics or syntax that causes them to be validated as true or false positives, and which should be considered a distinct variety of clone when measuring precision. While the clone pairs within a cluster are of the same variety, they may not be clones of each other, and a cluster may encompass multiple clone classes. The number of clusters captures the range of clone varieties the clone detector detects within the software system. The sizes of the clusters captures the distribution of these clone varieties in the software system (how common or uncommon). Precision can then be measured by validating a single randomly chosen exemplar clone pair from each cluster. This efficiently maximizes the variety of the clone pairs examined when measuring precision. It avoids duplicate manual validation efforts by considering each variety of clone only once, allowing the measurement of precision to scale better to a large collection of subject systems. Sampling clones across the clusters eliminates 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 in an experiment using the state of the art NiCad [18] as our subject clone detector and a portion of the Java [19] class library as our subject system. We vectorized the clone pairs using a custom document vectorization technique, and performed dimensionality reduction with principal component analysis [20]. We explored clustering solutions with both the k-means clustering algorithm [21] and by fitting Guassian mixture models using the expectation maximization algorithm [22]. We used the silhouette [23] metric to choose an optimal clustering solution, which we found to be produced by the k-means algorithm for 100 clusters. We manually inspected the clustering solution and found it was effective in separating the clone varieties, although not perfect (it is not realistic to expect a perfect clustering). We found that 76 clusters contained only a single variety of clones, although a few varieties were split across a small number of clusters, and 24 clusters contained multiple varieties of clones. A 100 clone pair sample produced by selecting one clone pair per cluster yields 76 distinct varieties of clones. This is over twice the variety of clones obtained by a random sampling of the same sample

size, and three times as efficient as the random sample size required to meet the same variety. Our technique reduces the bias due to the specific distribution of the clone varieties in the subject system under test by up to an order of magnitude per variety of clone. Our evaluation shows us that our technique can efficiently measure an accurate and generalized precision of a subject clone detection tool.

2. Clone Types and Varieties

Clones are *code fragments* within a software system that are similar, either syntactically and/or semantically. They are reported as either *clone pairs* or *clone classes*. Researchers agree on three primary clone types [1, 17]:

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 of code fragments that are similar, specified by tuple (f_1, f_2, ϕ), including the similar code fragments f_1 and f_2, and their clone type ϕ.

Clone Class: A set of code fragments that are similar. Specified by tuple $(f_1, f_2, ..., f_n, \phi)$. Each pair of distinct fragments is a clone pair: $(f_i, f_j, \phi),\ i, j \in 1..n,\ i \neq j$.

Type-1 Clone: Identical code fragments, except for differences in white space, layout and comments.

Type-2 Clone: Identical code fragments, except for differences in identifier names and literal values, in addition to Type-1 clone differences.

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.

Our goal in this paper is to measure a more accurate and generalized clone detection precision by validating a wide variety of clones detected by a tool without bias to any particular variety. We consider two clone pairs to be different clone varieties if they represent different cloning experiences from the perspective of clone validation. 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 true or false positives. This is different from clone types. When measuring precision we only want to validate a single exemplar from each clone variety because 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 (different distribution).

As an example, clone detectors often report simple constructors as clones. These are constructors that take a number of arguments and initialize member fields with their values. These clones may vary by length, or some may include a super() call, but in all cases validation depends on the decision of if two simple constructors

form a true clone. So we consider this a distinct variety of clone. Two clone pairs of this variety may not be clones of each other due to differences in the size of the constructors. Some other clone varieties we have observed in this study include: clones of methods that register action listeners, clones of equals() methods auto-generated by Eclipse IDE, clones of methods that implement buffer slicing, and so on.

We are not attempting to build a taxonomy of all varieties of clones. 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. As well as to measure the total variety of clones considered when measuring precision for a subject system or a set of subject systems.

3. 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.

3.1. *K-Means Clustering*

K-means [21] is an iterative clustering algorithm that aims to partition the data in a way that minimizes a loss function: the within-cluster sum of squares as shown in Eq. 1. 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. 2, is the preferred distance metric [26].

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

$$d(\vec{a}, \vec{b}) = 1 - cos(\theta) = 1 - \frac{\vec{a} \cdot \vec{b}}{||\vec{a}||\ ||\vec{b}||} \tag{2}$$

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 following algorithm:

(1) Assign each data point to its nearest cluster center using the chosen distance measure.
(2) Update the cluster centers to 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.

The algorithm is guaranteed to converge to a local optimum, although this may not be the global optimum. The clustering solution returned depends on the number of clusters and the initial cluster centers used.

3.2. *Initializing K-Means with K-Means++*

K-means++ [21] is an algorithm for choosing initial cluster centers for the k-means algorithm. 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 [21]. 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.

Each cluster center is chosen non-deterministically, while the weighted probability prefers to select the next center as a point further from the previously selected centers. It ensures the centers are well spread amongst the data.

3.3. *Gaussian Mixture Model (GMM)*

A Gaussian mixture model [27] is a probability distribution composed of the combination of a number of weighted individual Gaussian distributions. It can be used to model data where the population is believed to contain a number of sub-populations, even when these sub-populations are not known. By learning the parameters of the individual Gaussian distributions, the data can be clustered into the sub-populations they are most likely to originate from.

To solve the problem of clone clustering, we consider the clones to have been drawn from a mixture of k multivariate Gaussian distributions. Each Gaussian distribution is a sub-population, or cluster of the clones. We assume each clone was drawn from just one of the individual Gaussian distributions, and the drawing of each clone is independent of the drawing of each of the other clones.

The i^{th} Gaussian is shown in Eq. 3 for the representation of a clone, $\vec{x}$, as a d-dimensional vector of real numbers. The mixture of k Gaussian distributions is shown in Eq. 4, where α_i is the weighting of the i^{th} Gaussian distribution in the mixture, and $\sum_{i=1}^{k} \alpha_i = 1$.

To complete this model we need to learn the model parameters, Θ. Where Θ includes the number of Gaussian distributions, k, their weightings in the mixture $\vec{\alpha} = \{\alpha_1, \alpha_2, ..., \alpha_k\}$, and the parameters of each individual Gaussian distribution, $\theta_i = \{\vec{\mu_i}, \Sigma_i\}$. Given a known number of Gaussian distributions (clusters), k, the weightings, means and covariances of the Gaussian distributions can be learned using the Expectation Maximization algorithm, which is discussed in the following subsection.

$$p_i(\vec{x}|\mu_i, \Sigma_i) = \frac{1}{(2\pi)^{d/2}\,|\Sigma_i|^2} e^{-\frac{1}{2}(\vec{x}-\vec{\mu_i})^t \Sigma_i^{-1}(\vec{x}-\vec{\mu_i})} \tag{3}$$

$$p(\vec{x}|\Theta) = \sum_{i=1}^{k} \alpha_i p_i(\vec{x}|\mu_i, \Sigma_i) \tag{4}$$

The clones in the dataset are then assigned to the Gaussian distribution in the mixture most responsible for generating it. In other words, the cluster the clone is most likely to belong to. The **responsibility** cluster i takes for clone c is shown in Eq. 5, where $\sum_{i=1}^{k} r_{ci} = 1$.

$$r_{ci} = p(c_i = 1|\vec{x}_c, \Theta) = \frac{\alpha_i \cdot p_i(\vec{x}_c|\theta_i)}{\sum_{m=1}^{k} \alpha_m \cdot p_m(\vec{x}_c|\theta_m)} \tag{5}$$

3.4. *Expectation Maximization (EM) Clustering*

Given a known number of mixture components, k, the Expectation Maximization algorithm [27] can approximate the **maximum likelihood estimate** (MLE) of the parameters of each Gaussian component of the mixture model. This is the weighting α_i, mean μ_i, and covariance Σ_i of each of the k Gaussian distributions, $i = (1, 2, ..., k)$.

The algorithm requires initialization with the number of Gaussian distributions in the mixture k, an initial weighting of their contribution $\vec{\alpha}$, and an estimate of the mean and covariance, $\theta_i = \{\vec{\mu_i}, \Sigma_i\}$, of each Gaussian distribution. The initial weightings and Gaussian distribution component parameters can be chosen randomly, or from a k-means clustering solution (discussed in the next subsection).

The algorithm then iteratively updates $\Theta = \{\vec{\alpha}, \theta\}$ until convergence of the log-likelihood of the parameters, or some maximum number of iterations. Only the k parameter is kept fixed during the algorithm. Each iteration is guaranteed to increase the likelihood of the parameters given the data. However, it may not find *the* Θ_{MLE}, but instead return a local maximum. Each iteration involves an **expectation (E)** step and a **maximization (M)** step.

E-Step. In the expectation step, the responsibility, r_{ci}, each Gaussian distribution (cluster), c_i, takes for generating each clone, $\vec{x_c}$, in the dataset is calculated. This is equivalent to calculating the expectation of the log-likelihood of the posterior distribution given the current estimates of the parameters [27].

M-Step. In the maximization step, Θ is updated to maximize the expected log-likelihood using a number of derived equations [27]. Specifically, new weightings and the Gaussian distribution component parameters (mean and covariance) are updated as shown in Eq. 6 and Eq. 7, where N is the number of data points.

$$\alpha_i' = \frac{N_i}{N} \qquad \vec{\mu'}_i = \frac{1}{N_i}\sum_{c=1}^{N} r_{ci} \cdot \vec{x}_c \qquad N_i = \sum_{c=1}^{N} r_{ci}. \tag{6}$$

$$\Sigma_i' = \frac{1}{N_i}\sum_{c=1}^{N} r_{ci} \cdot (\vec{x}_c - \vec{\mu'}_i)(\vec{x}_c - \vec{\mu'}_i)^t \tag{7}$$

Convergence is detected when the log-likelihood of Θ, shown in Eq. 8, no longer increases by an appreciable amount.

$$\log l(\Theta) = \sum_{c=1}^{N} \log p(\vec{x}_c|\Theta) = \sum_{c=1}^{N} \Big(\log \sum_{i=1}^{K} \alpha_i \cdot p_i(\vec{x}_c|\theta_i, \Sigma_i) \Big) \tag{8}$$

While the expectation maximization algorithm can be initialized with random parameters, it is ideal to start with good estimates. Assuming a chosen number of clusters, k, initial values for the Gaussian can be derived from a clustering solution produced by a simpler clustering algorithm. Specifically, we target the use of k-means to provide an initial clustering of the clones to initialize the Gaussian mixture model for the expectation maximization algorithm.

3.5. *Initializing EM using K-means*

The results of the k-means algorithm can be used to initialize a Gaussian mixture model for the expectation maximization algorithm. The output of the k-means algorithm is the clones labeled with their cluster id. For each k-means cluster, m_k, a Gaussian distribution is initialized in the mixture with parameters shown in Eq. 9. The initial weightings of the Gaussian components can be uniform, or initialized to the ratio of the dataset encapsulated in their corresponding k-means clusters. The mean of the Gaussian is set as the centroid (average) of the cluster, and the covariance as the covariance of the cluster's data-points (clones).

$$\alpha_i = \frac{N_i}{N} \quad \vec{\mu_i} = \frac{\sum_{x \epsilon m_k} x}{N_k} \quad \Sigma_{ab}^i = cov(X_a, X_b) \tag{9}$$

3.6. *Silhouette Metric*

The silhouette metric [23, 24] 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. 10, 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. 2).

$$s(i) = \frac{b(i) - a(i)}{max\{a(i), b(i)\}} \tag{10}$$

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 (or an individual cluster) 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 we must control for. If a cluster is created per data point (or per unique data point, if the dataset contains duplicates), 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.

3.7. *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' [25] 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 drops. This estimates the natural number of clusters as the point where adding additional clusters does not significantly improve the quality

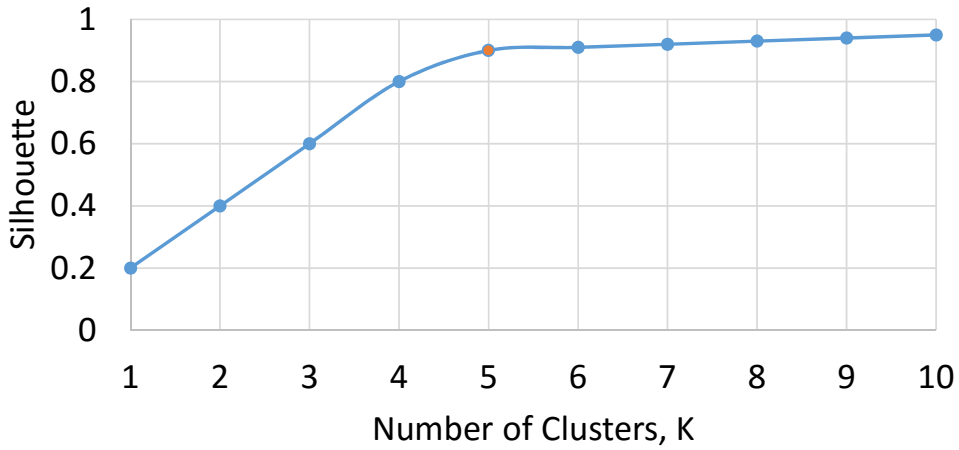

Fig. 1: Example of the Elbow Method

of the clustering solution. A contrived example of the elbow method is shown in Figure 1, where the elbow occurs at five clusters.

3.8. *Principal Component Analysis (PCA)*

Principle component analysis [20] 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. For example, T may be chosen to preserve 90% of the data's total variance.

4. Measuring Precision with Clone Clusters

In this section, we overview how the precision of a clone detector can be measured using a clone clustering solution. The subject clone detector is executed for a subject software system (or collection of software systems), and the detected clone pairs are vectorized and clustered by a clustering algorithm (we explore this in detail in Section 5). This labels each clone pair with a value $[1, k]$, where k is the number of clusters. Note that we are clustering clone pairs as a whole unit, not their individual code fragments. We cluster clone pairs rather than clone classes because clone detectors might might mix true and false positives and different varieties of clone pairs within the same class, because validating (especially large) Type-2 and Type-3 clone classes can be unwieldy, and because not all clone detectors report clone classes. The number of clusters in the clustering solution captures the range

of the clone varieties detected in the subject system(s). The sizes of the clusters indicates the distribution (how common/uncommon) of these clones varieties in the tool's detection report.

The goal is to cluster the similar clone pairs together such that each cluster represents a distinct variety of clones, as described in Section 2. Although all the clone pairs within a cluster are of the same variety, the cluster may contain multiple clone classes, and the clones within a cluster may not be clones of each other. To measure precision, an exemplar clone pair is randomly selected from each cluster and manually judged as a true or false positive by a clone expert(s). Each exemplar represents the clone variety of its originating cluster. The validated exemplars and their originating clusters can be used to measure precision. This technique maximizes the variety of clones considered when measuring precision, while minimizing the number of clones that need to be manually validated. This procedure could be executed for a number of diverse subject software systems, with precision measured for each system individually and than averaged. Alternatively, the clones detected across a variety of software systems could be combined before clustering. This would minimize the sample size while maximizing the variety of detected clones inspected for the subject tool across a community of software systems.

Precision can then be measured in multiple ways. It can be measured as the ratio of the exemplar clone pairs that are true positives. This gives each variety of clone a uniform weighting in the measurement of precision, regardless of how common (cluster size) each variety of clone is in the target subject system(s). This is a generalized measure of precision, as it normalizes for the distribution of the varieties of clones in the specific subject system(s) used. This allows the precision to remain true for an arbitrary software system where the distribution of clone varieties are different. Alternatively, the non-generalized precision for the specific subject system(s) can be measured by weighting the contribution of the exemplars by the size of their corresponding clusters. This is similar to measuring precision by a random sampling where more common clones have a stronger impact on the precision. However, a pure random sampling requires a large sample to ensure the population is sufficiently represented. By clustering the clones we can ensure the variety of clones has been examined while minimizing the number of clones that need to be manually validated.

The above procedure assumes a perfect clustering, where every cluster contains exactly one variety of clones, and each cluster has a different clone variety than the others. However, it is unrealistic to expect a perfect clustering. It is difficult to capture the semantic concept of similar clones, from the perspective of clone validation, numerically for a clustering algorithm. It is also challenging to learn the number of clone varieties (clusters) in a clone dataset. The clustering algorithms themselves also have limitations. Specifically, a variety of clones may be split across multiple clusters, or a cluster may contain multiple varieties of clones. However, despite this, we find we are able to achieve a good result, and a high-quality precision

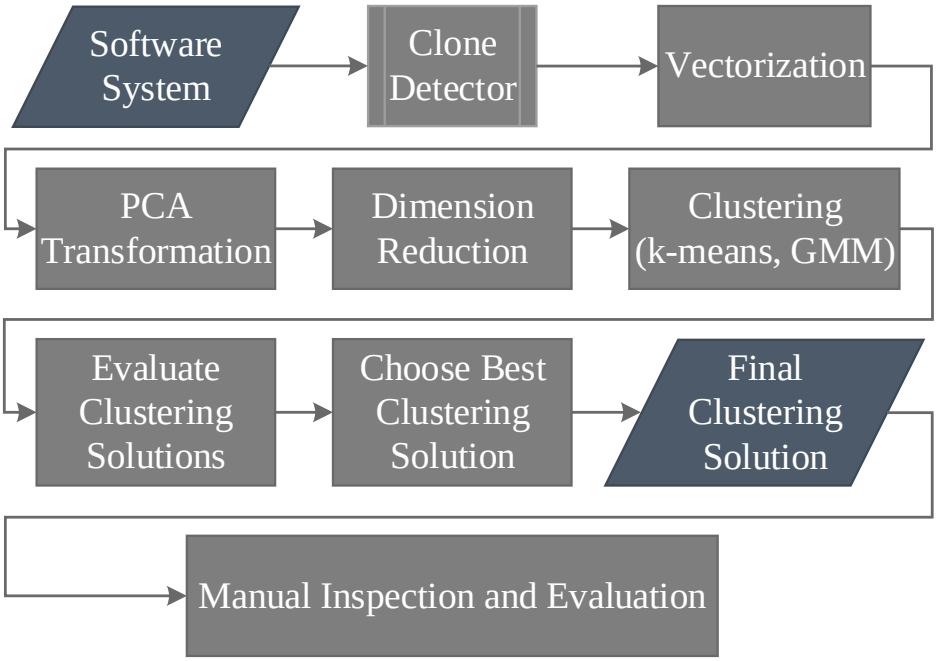

Fig. 2: Overview of Experimental Procedure

measurement. In the remainder of the paper we explore clone clustering (Section 5), and we evaluate how well this enables the measurement of precision as proposed in this procedure (Section 6).

5. Experiment

In this section we experiment with clustering clones using the k-means and Guassian Mixture Model (GMM) clustering algorithms for the purpose of measuring precision as described in Section 4. Our experimental procedure is summarized in Figure 2. We begin by executing a clone detection tool for a software system. For this experiment, we use the NiCad clone detector as our subject clone detector, and a package of the Java class library as our subject software system. We then vectorize the clone pairs into real-valued vectors using an adaptation of standard document vectorization, specifically targeting the properties of clone pairs as documents.

The clone vectors are in a very high dimensionality, and clustering algorithms are known to perform poorly in high dimensions, so we perform dimensionality reduction with principle component analysis (Section 3.8). First we rotate the vectors onto their principle components, and then keep only the Top-T dimensions that capture a target total percentage of the original data variance. Using the k-means algorithm and our clustering quality metrics (Section 3.6), we explore different val-

ues of T while varying the number of clusters, k. We choose the best value of T as the largest reduction in dimensionality that still indicates a natural clustering point using the elbow method (Section 3.7).

We then cluster the clones using the reduced clone vectors using both the k-means (Section 3.1-3.2) and GMM (Section 3.3-3.5) algorithms. We execute both algorithms for a range of number of clusters, k, and select their best clustering solutions using the silhouette and percentage of singleton clusters metrics (Section 3.6), and the knee method (Section 3.7). We initialize the k-means algorithm using the k-means++ algorithm, and initialize the GMM algorithm using the corresponding k-means clustering solutions.

We then compare the algorithms using the metrics to determine which provided us the best clustering solution. We manually inspect this final clustering solution to determine how well it has clustered the clones for use with our procedure for measuring precision (Section 4) using a clone clustering.

In the following subsections we go over each step of the experiment in detail.

5.1. *Subject Software System*

We use the source files under the top-level 'java' package of the Java 8 class library as our subject system. Packages of the Java class library are a common target in clone studies [17, 8]. The 'java' package includes a diverse collection of functionalities, and therefore should contain a significant variety of clones. As a standard library, we also expect it to repeat various programming patterns and practices, and therefore should also contain a number of similar clones. The 'java' package contains 1868 Java source files encompassing 291 thousand lines of Java code.

Previously, we propose that clustering for measuring precision could be applied to clones detected in a single subject system or to the combination of clones detected in a variety of software systems. This subject system represents both cases well. The 'java' package is itself a complete subject system, while its 56 sub-packages are also complete subject systems. In this way we are investigating clone clustering for many datasets in a cross-project clone detection scenario. These properties make this an ideal subject system for our study. Of course, our approach could be executed for any subject system and programming language.

5.2. *Subject Clone Detector*

We use NiCad [18] as our subject clone detector. NiCad is a popular and state of the art clone detector with strong recall [28, 8]. It is a hybrid abstract-syntax-tree (AST) and text-based clone detector. It uses TXL to parse the source files, extract the code fragments, and perform source normalizations to improve clone detection. It supports clone detection in Java, C, C# and python software systems at the block, function and file granularity. It uses a strict pretty-printing, which normalizes spacing and line breaks and removes comments. This exposes Type-1 clones as textually identical code fragments. It includes identifier renaming and

literal value normalizations, which expose Type-2 clones as textually identical code fragments. Type-3 clones are detected by an optimized longest common subsequence algorithm, along with a maximum clone dissimilarity threshold. NiCad supports additional TXL-based source transformations, and extension to additional source languages and granularities, by a plugin architecture.

We executed NiCad for the detection of function granularity clones with no more than 30% dissimilarity after blind identifier normalization and literal value abstraction. We use a minimum clone size of 6 lines, which is common in clone detection tool evaluation [17, 28]. In total, 14,076 clone pairs were detected in the subject system. We intentionally used a generous configuration to ensure we detect both true and false clones, for the best test of our clustering procedure.

5.3. *Clone Vectorization*

Clustering algorithms require the data to be represented by a n by d matrix of real or discrete numerical values. Where n is the number of data points (clone pairs), and d is the number of features per data point. In order to cluster the clones we need to convert them into representative real-valued vectors in d dimensions. To do this we use standard document vectorization [26].

The most common document vectorization technique is to consider a document as a bag-of-terms, and represent it as a vector of term-frequencies in term-space. The vector is typically weighted by the inverse-document-frequencies of the terms, under the assumption that frequent terms have less discriminating power when clustering documents. A document has the vector form shown in Eq. 11, 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 number of documents, and d unique terms occur across all of the documents.

$$document = \{tf_1 \log \frac{n}{df_1}, tf_2 \log \frac{n}{df_2}, ..., tf_d \log \frac{n}{df_d}\} \tag{11}$$

Clones are pairs of similar code fragments. To apply the document vectorization technique we need to convert the clone pairs into a single document representation, and choose a term definition. A natural choice from the clone definitions (Section 2) is to represent a clone pair as a document of the statement-level cloned code patterns (as terms) shared by its code fragments. From the definitions, the most important aspect is the code shared by the code fragments. The clone-type definitions indicate that clones may contain differences in identifiers and literals, so we normalize these differences, turning the code fragments into code patterns over arbitrary identifier names and literal values.

We choose to capture code patterns at the statement-level as statements capture semantically complete steps. Java has a fairly concise syntax, so it may be trivial for clones to share normalized statement code patterns. Instead, we capture higher-level code patterns by considering sequences of multiple statement as code patterns using

n-grams. The goal is to cluster clone pairs that have similar high-level syntax and semantics in their cloned-code from the perspective of clone validation. For this, we ignore the non-cloned code and the minor lower-level syntactical similarities. The full vectorization process is described in the following subsections. While we target Java for this experiment, this technique is applicable for any programming language.

While this process shares some processes with clone detectors – source normalization and capturing of cloned code regions – this is different from clone detection. Our goal here is to 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. Therefore, this process is designed to consider only the significant cloned features of the clone pairs, not the fine-grained cloned and non-cloned features that define variance within a clone variety.

5.3.1. *Extracting Shared Code Patterns*

Before vectorization, we must convert the clones into their single-document representations by extracting their shared code patterns. We begin by extracting the clone pair's code fragments from the software system. An example clone pair is shown in Figure 3a. Next we convert the code fragments into single-statement code patterns as shown in Figure 3b. This is achieved using TXL-based [29] source-code normalizations. First we use a strict pretty-printing, which reformats the code to one statement per line. Then we replace all identifier names with the token 'X' and all literal values (strings, numerical, boolean, null) with the token '0'.

We capture the higher-level code patterns in the code fragments by applying an n-gram transformation across the single-statement code patterns, as shown in Figure 3c for $n = 3$. An advantage of n-grams is it preserves some of the statement sequence information that would otherwise be lost in the bag-of-terms representation. The goal is to capture just the high-level cloned code patterns between the clone's code fragments, so that the clustering algorithm groups clones based on similarities in their major syntactical and semantic features. This is a compromise between capturing high-level semantics in the code, and being mindful that Type-3 clones contain statement-level gaps that may split sequences of shared code statements.

We now treat each code fragment as a bag of (N-statement) code patterns. We extract the shared code patterns by computing the intersection of the two bags. With the 3-gram representation, only the shared code patterns that are three statements or longer are kept. This prevents clones that share only trivial cloned code patterns from being clustered together. Duplicate 3-grams are allowed in the bags, and the number of duplicates shared is computed by the intersection. The single-document representation of the clone is then a sequence of the shared code patterns, as shown in Figure 3d. The order of this sequence is not important as the

```
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);
}
```

(a) Original Clone Pair

```
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);
}
```

(b) 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(); 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); }
```

(c) Higher Level Code Patterns Using 3-Gram Transformation

```
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();
```

(d) Shared Higher Level Code Patterns

Fig. 3: Clone-Pair to Single-Document Conversion

next step is vectorization.

5.3.2. *Vectorization*

The single-document representations of the clones are vectorized using the N-gram code patterns as terms, as described in Eq. 11. For our subject system, we found a significant number of the terms only appear in a single clone. The inverse-document-frequency puts a significant weight on these dimensions, while they do not help cluster these clones with other similar clones, and increases the risk of single-point clusters, which are undesirable. We therefore drop these terms from the vectors. This reduces the overall dimensionality of the data, which can improve clustering performance [30]. We normalize the vectors by their length in order to ignore differ-

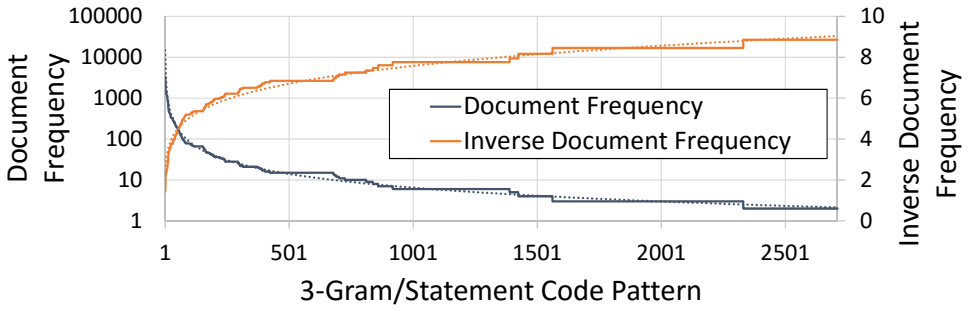

Fig. 4: Document Frequency of 3-Grams in the Dataset

ences in clone (shared code pattern) size. The clones are output as a n by d matrix ready for clustering.

5.3.3. *Applying to the Dataset*

We vectorized the clone pairs NiCad detected in our subject system using the above process. We used 3-gram (statement) code pattern terms for document vectorization with term-frequency inverse-document-frequency (tf-idf). The vectors are over 2711 term dimensions. This is a reduction from 7719 term dimensions after removing the singleton terms (those that appear in only a single clone document). Figure 4 plots the document frequencies and inverse document frequency weights of the non-singleton terms in order of decreasing document frequency. The document frequency very closely matches a power-law distribution. Most of the terms appear in 2-100 of the clone documents. Only 71 terms appear in more than 100 clone documents, with the most frequent term appearing in 3350 clone documents. The inverse-document-frequency weighting in the vectorization accounts for this disparity in common and rare code patterns, and prevents clusters from being trivially created by only the common terms. It is the sharing of rare terms that is most likely to indicate similar clones.

The clone vectors are very sparse. On average, a clone vector has a non-zero magnitude in just 4.2 dimensions. Across all of the clone vectors, there are only 59,947 non-zero real values. While the dataset includes 14,076 vectorized clones, only 1422 of these vectors are unique. Clone pairs with the same vector representation are not necessarily identical clones, but their single-document representation has the same terms (3-gram code patterns) in the same ratios. Both clones have code fragments that share the same 3-statement code patterns in the same ratios, ignoring differences in clone size.

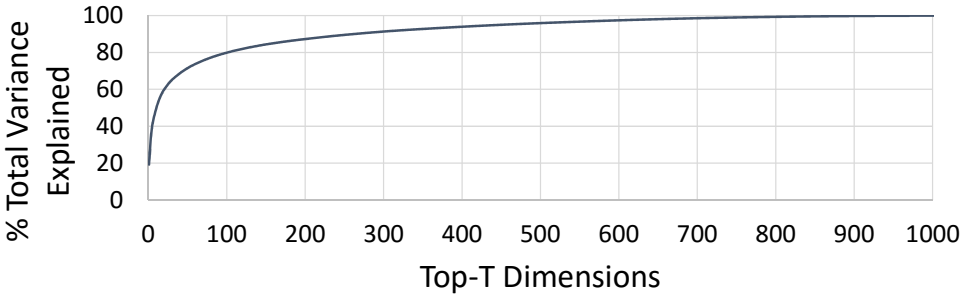

Fig. 5: Total Variance Explained by PCA Reduction

5.4. *Dimensionality Reduction*

The vectorized clones are in a very high dimensionality. Clustering algorithms are known to perform poorly in high dimensions [30]. In high dimensions, distance metrics begin to return similar values for any arbitrary pair of data vectors due to high degree of orthogonality.

Dimensional reduction can be achieved using principal component analysis (Section 3.8). The clone vectors are rotated onto a linearly uncorrelated principal component basis. The principal components are in decreasing order of the original data variance they explain. Reduction is achieved by keeping only the top-T principal components (dimensions) that preserve a target total data variance explained.

We used MatLab [31] to perform PCA on the data. This returns the rotated data as well as the variance explained by each principal component. Figure 5 shows the total variance explained by the Top-T principle components (dimensions). Essentially 100% of the variance is explained by the first 1000 of 2711 principle components. However, the majority of the total variance is explained by only the first 15-250 principal components. Ideally, we want a significant reduction from the original 2711 term dimensions.

We explore values of T for preserving 50-90% of the variance in the data. This corresponds to convenient T values of 15 (56%), 25 (63%), 50 (71%), 100 (80%), and 250 (90%). We execute an initial k-means clustering (with k-means++ initialization and using the cosine distance metric) for each reduction for a range of k from 20 to 1300. We measure the silhouette of each clustering solution, and plot this for each reduction and across k in Figure 6. We want to choose the reduction which best indicates a natural clustering point, as determined by the elbow method (Section 3.7), while optimizing the silhouette at this clustering point.

A reduction to the first 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 have either multiple and/or unclear elbows. It achieves a higher silhouette than keeping more dimensions (250), while having comparable silhouette to the further reductions (15-50). So we choose to

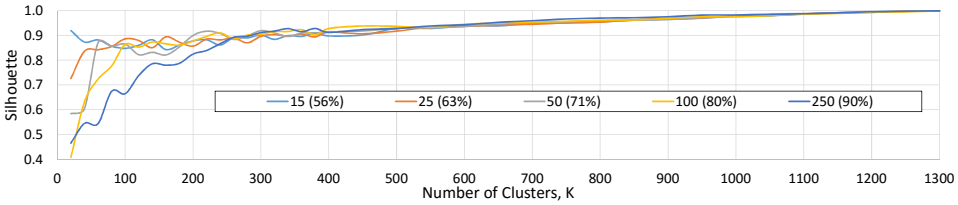

Fig. 6: Silhouette of K-Means Solutions Across k for Different PCA Dimensionality Reductions

reduce the PCA-transformed dataset to the first 100 principal components for use with k-means clustering. This also applies to the GMM clustering, which we initialize using k-means clustering.

5.5. *K-Means Clustering*

We now explore k-means clustering solutions for a range of k from 20 to 1300 on the clone vectors after PCA dimension reduction keeping the top-100 principal components. We use MatLab's [31] k-means implementation with k-means++ initialization and cosine distance metric. We use an increment of 20 in the range 20-400 where the elbow is observed, and an increment of 50 from 400-1300. It was not executed for every k in this range due to execution time considerations.

For each clustering solution, we measured its silhouette and percentage of singleton clusters, as described in Section 3.6. These are shown across k in Figure 7. The silhouette increases rapidly as the number of clusters increases to 100. Afterwards, the silhouette increases slowly to near 1.0 as the number of clusters increases to 1300. After 1300 clusters, the k-means algorithm became unstable, splitting duplicate clone vectors into different clusters, which cause the silhouette measure to become undefined. The 'elbow method' (Section 3.7) on the silhouette indicates that 100 clusters is the natural number of clusters for this data. This is the point where the improvement in clustering quality (silhouette) by adding additional clusters sharply declines.

This is supported by the 'percentage of singleton' clusters measure. Remember (Section 3.6) that the silhouette metric has the flaw where it trivially returns a perfectly clustering, 1.0, when there is a cluster for each unique data point (allowing duplicates). 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 strongly due to the increase in singleton clusters. 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 the best k-means solution is achieved at $k = 100$ clusters. This clustering solution has a silhouette of 0.86 with

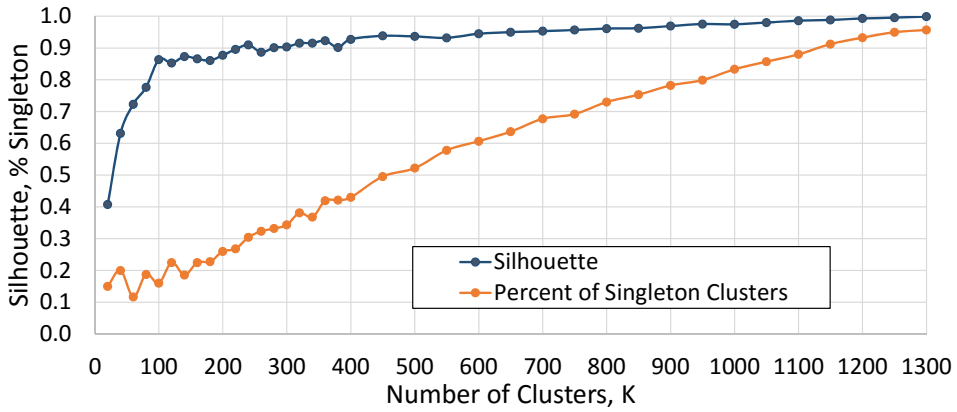

Fig. 7: Evaluation of K-Means Clustering Solutions

16% singleton clusters.

5.6. *GMM Clustering*

We now explore GMM clustering solutions for a range of k from 20 to 400. Clustering is performed by initializing a mixture model of k Gaussian distributions (Section 3.3) and fitting it using the expectation maximization algorithm (Section 3.4). The data points (clones) are assigned to the Gaussian component (cluster) most likely to have generated it. Fitting the Gaussian mixture model was accomplished using MatLab.

The Gaussian mixture models were initialized using their corresponding k-means clustering solution as described in Section 3.5, using an uniform initial weighting. The EM algorithm is very sensitive to initial conditions, so we use the k-means solution to provide a good starting condition. This can be seen as using a GMM clustering to refine the k-means solutions by modeling the covariance in the clustered data. A regularization value of $1x10^{-6}$ was used to prevent ill-fitting covariance matrices. The covariance matrices were restricted to diagonal matrices to reduce the number of free parameters.

Restricting to diagonal covariance matrices greatly reduces the number of free parameters in our case. It is a requirement as otherwise there is not enough data to properly fit the Gaussian mixture models, although diagonal covariance matrices somewhat reduces the power of the model. After PCA dimensionality reduction our data is in 100 dimensions. Assuming a natural clustering point of around 100 clusters (as found with k-means), full covariance matrices would require one million free parameters. An additional 10,100 free parameters are required for the Gaussian means and mixture weightings. However, we have only 14,072 data vectors spanning 1.41 million real-values. The sparsity of the data reduce this to 59,947 non-zero real-values. This is insufficient to fit on the order of one million free parameters. Diagonal matrices reduces the free parameters from the covariance matrices to 10,000, or

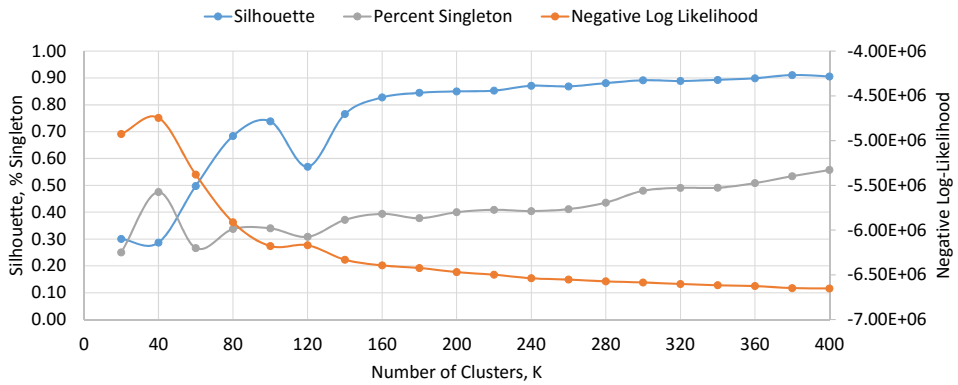

Fig. 8: Evaluation of Gaussian Mixture Model Clustering Solutions

20,100 free parameters in total. With diagonal covariance matrices the data at least exceeds the number of free parameters. This is the challenge of using Gaussian mixture model to cluster data with a large number of natural clusters and high dimensionality.

For each GMM clustering solution, we measured their silhouette and percentage of singleton clusters. Since this is a probabilistic model, we were also able to measure the negative log-likelihood of the clustering solutions. These are plotted against number of clusters (Guassian components), k, in Figure 8. Like the k-means solutions, the silhouette has most of its gains increasing the number of clusters from 20 to 100. Afterwards, the silhouette slowly increases towards 1.0 as the percentage of singleton clusters increases. We stopped at 400 clusters with GMM as the number of free parameters begins to overwhelm the data.

There is a strange feature in the silhouette plot at 120 clusters where the silhouette drops dramatically before recovering at 140 clusters. This feature makes it more challenging to apply the 'knee method' in choosing a natural number of clusters. The knees in the plot appear to be at 100 and 140 or 160 clusters. If we assume the 120 clusters solution is an anomaly and smooth over it, than 100 clusters would be the single knee. This is supported by the plot of the negative log likelihood, where the significant (negative) increase in log-likelihood occurs up to 100 clusters, with weak increase afterwards.

So the best GMM clustering solution is for 100 clusters. This achieves a silhouette of 0.74 with 34% singleton clusters. We compare this against the k-means clustering solutions in the next section.

5.7. *Comparing Clustering Solutions*

In this section we compare the k-means and GMM clustering solutions to determine which provides the better clustering. From both algorithms, the natural clustering

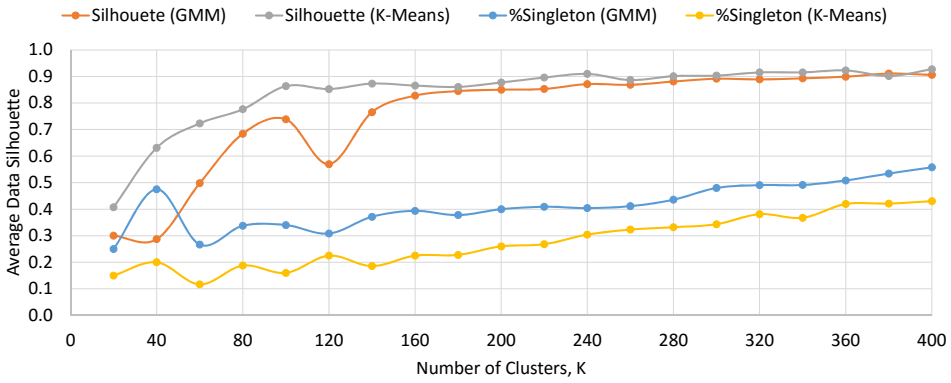

Fig. 9: k-means vs. GMM Clustering Solutions

point appears to be around 100 clusters. At 100 clusters, k-means achieves a clustering solution with a silhouette of 0.86 and a low percentage of singleton clusters, 16%. While GMM achieves a silhouette of 0.74 with 34% singleton clusters. Both algorithms achieve a good silhouette, although k-means achieves a higher silhouette. The k-means solution has a much lower percentage of singleton clusters, 16% versus GMM's 34%.

To get a better idea of how these algorithms compare, we compare their silhouette and percentage of singleton clusters across the range of k from 20 to 400, which is plotted in Figure 9. As can be seen, the k-means algorithm consistently provides clustering solutions that achieve higher silhouette with fewer singleton clusters. The k-means solutions is also free of anomalies, such as the one occurring at 120 clusters with GMM clustering. Both algorithms have a similar phenomenon where the silhouette slowly approaches 1.0 as the percentage of singleton clusters increases.

Overall, k-means provides better clustering solutions for our clone data. In general, it is expected that GMM clustering can provide a better clustering solution since it models the covariance of the clusters. There are a number of possible reasons why it is performing worse than k-means in this case. One possibility is that the distribution of the clone data cannot be approximated by a mixture of Gaussian distributions. Another likely possibility is there is simply not enough data to approximate the parameters of the Gaussian mixture model. With a natural clustering point of 100 clusters, the Gaussian mixture model (limited to diagonal covariances) requires 20,100 free parameters, whereas the dataset contains only 14,072 sparse data vectors totaling 59,947 non-zero data values. We therefore use the best k-means clustering solution as our final clustering solution.

5.8. *Final Clustering Solution*

In the previous sections, we found that k-means provides the best clustering solution, with a natural clustering point at 100 clusters with dimensionality reduction

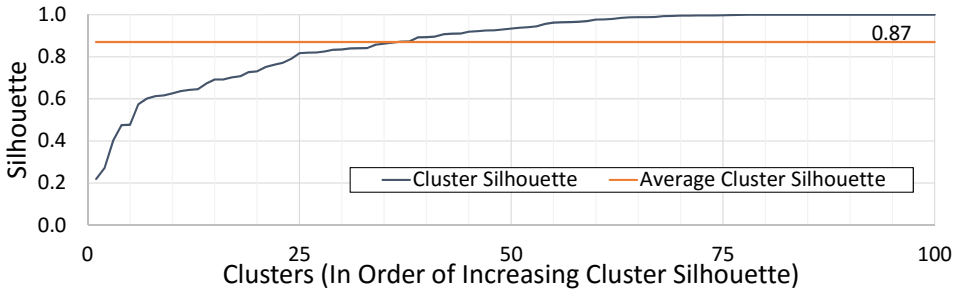

Fig. 10: Distribution of Cluster Silhouette

to the top-100 dimensions after principle component analysis. For our final clustering solution, we re-executed the k-means algorithm for 100 clusters with the same parameters (k-means++ initialization, cosine distance) except with 10 replications. This means that the algorithm is executed ten times with different initial conditions, and the solution with the lowest total sums of squares is chosen as the optimal solution. This final clustering solution has a silhouette of 0.87 with 19% singleton clusters. In this section we measure properties of the final clustering solution.

While we measured the overall silhouette of the clustering solution (0.87), also of interest is the silhouette of the individual clusters. The silhouette of an individual cluster is measured as the average silhouette of only the data points in that cluster. The distribution of the individual cluster silhouettes is plotted in Figure 10 in order of increasing cluster silhouette. 75% of the clusters have an excellent silhouette, greater than 0.80. Of the remaining 25% below 0.80, only 5% have a silhouette less than 0.5, with one cluster having the worst silhouette of 0.2. The average cluster silhouette across these 100 clusters is 0.87, so the majority of the clusters are of high quality, as judged by the silhouette metric.

Figure 11 plots the distribution of cluster sizes in order of increasing cluster size, along with their cumulative ratio of the dataset. The clusters range from 15 to 2540 clone pairs in size. The majority of the clones are in the larger clusters. Figure 12 bins the clusters into convenient regions. The majority of the clusters (71%) contain only 15-99 clones and span only 24% of the clones, while 28% of the clones are contained within two very large (1440 and 2540 clones) clusters. The remaining 48% of the dataset occurs in clusters of sizes 101 to 702 clones, although the majority of these occur in the 100-194 range.

This confirms our knowledge of the distribution of clone variety in a software system. Given that each cluster groups similar clones, 71% of the variety makes up only 24% of the dataset. A pure random sampling of the clones will be biased by the clones in the larger clusters because they make up a more significant ratio of the total dataset. For example, 28% of a pure random sample should be from the two largest clusters. This is why we need to sample the clones per cluster to ensure

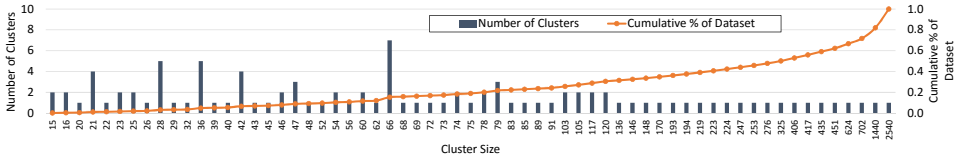

Fig. 11: Cluster Size Distribution

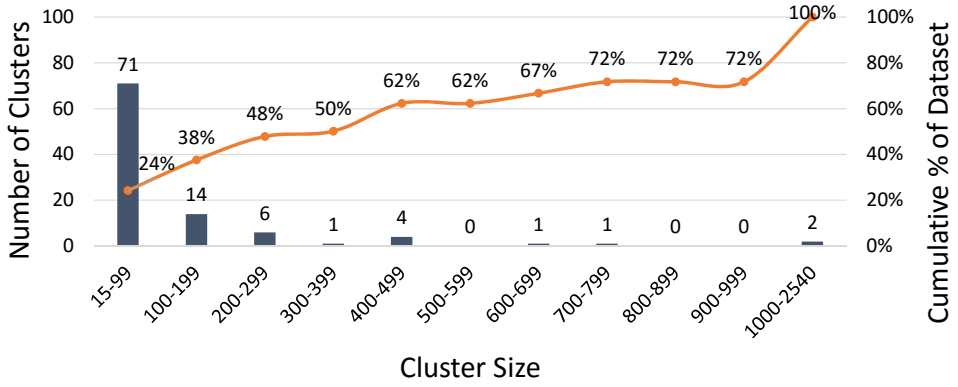

Fig. 12: Binned Cluster Size Distribution

maximum clone variety is captured in a minimum sample size.

5.9. *Manual Inspection*

We manually inspected the final clustering solution to see how well it meets our needs for measuring precision as previously outlined in Section 4. The goal was to produce a clustering of clone pairs where distinct varieties of clones are placed in their own cluster, and where all of the clone pairs in a cluster can be represented by a single randomly chosen exemplar from that cluster when measuring precision. Manual inspection of the clusters was done for evaluation only, and is not a required component of the procedure for measuring precision. During this inspection we judged the clusters as strong or weak. A "strong" cluster is one that contains only a single variety of clones, and which can be represented by a single randomly chosen exemplar from the cluster. A "weak" cluster contains multiple varieties of clones, and cannot be fully represented by a single exemplar. In total we found 76 strong clusters and 24 weak clusters.

When judging the clusters as strong or weak, i.e. if they contain a single or multiple varieties of clones (Section 2), 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

clone. A particular clone variety may span multiple clone types. A few examples of 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, in which case it would be unlikely to choose an outlier as the exemplar of the cluster.

In our approach (Section 4), precision is measured by validating a random exemplar clone from each cluster. Since the strong clusters contain clones of a single clone variety, and we judged clone variety based on the aspects that affect clone validation, it is safe to extrapolate the validation of the exemplar as a true or false positive to all clone pairs within the strong cluster. We now provide some detailed observations about the strong and weak clusters.

Strong Clusters: While inspecting the clusters we noticed that some of the strong clusters should ideally be merged. For example, there were multiple clusters containing clones of functions implementing object cloning using the same code pattern. These clusters were split by the k-means algorithm due to some differences in the code patterns their clones shared. This means that some of the exemplars from the strong clusters will be similar. We selected a random exemplar from each of the strong clusters and manually grouped those of the same variety of clone.

In total, the 76 strong clusters yielded 56 distinct clone varieties. Eleven clone varieties were split into multiple clusters. The majority of these (six) were split into two clusters, while two were split into three, two into four and one into five clusters. While the splitting of a clone variety into multiple strong clusters is not a problem in terms of missing a variety of clones, it does needlessly increase the number exemplars to be examined, and adds a small bias when a generalized precision is measured by giving each cluster equal weighting (Section 4). In this case it is minor, affecting only 11 of the 56 (20%) clone varieties found in the strong clusters.

Weak Clusters: The weak clusters are those that contain multiple clone varieties, where a single exemplar cannot represent the whole cluster when measuring precision. This does not mean the clones within the cluster do not share features, nor that they are all completely disparate, the cluster just simply cannot be represented by only a single exemplar. 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. These hidden strong clusters have been merged by the algorithm due to sharing a code pattern or sharing linearly correlated code patterns (due to rotation onto principal components).

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. 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. In the next section we provide a full quantitative analysis of the performance of our technique and compare it against the traditional random sampling approach.

Summary: 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. So 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, as we found in Section 5.9. This is an efficiency of 76% in terms of clone variety within the inspected clone pairs.

6. 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 final clustering solution (k-means, 100 clusters) 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. 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

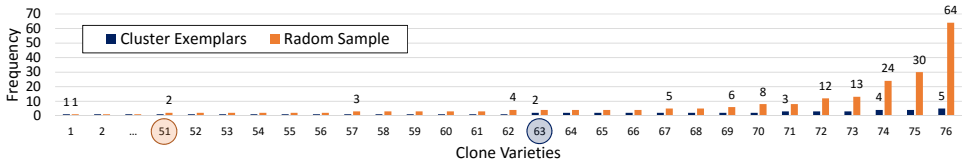

Fig. 13: Distribution of the Varieties of Clones Sampled - Our Cluster-Based Approach vs Random Sampling

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 13. 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 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 specific precision of a particular subject system, it is not effective in measuring a generalized precision. Our cluster-based technique excels in measuring an accurate and generalized precision. It can also be used to measure the specific precision for a particular subject system by weighting each exemplar by the size of the cluster it was drawn from. Our technique requires a smaller sample size, therefore less efforts, than random sampling.

7. Case Study: NiCad

We now measure the precision of NiCad using the clustering solution. We randomly selected one exemplar clone pair per cluster (100 clones) for manual validation. We judged a clone pair as true positive if its code fragments shared syntax and/or semantics, in addition to it being useful or important to a software maintenance

and evolution task or concern. We judged a clone pair as false positive if it did not share syntax or semantics, or if it only trivially did and would't be useful to a developer. We measured a generalized precision of 74% by giving each exemplar clone equal weighting. We measure a specific precision of 54% for the subject system by weighting the contribution of each exemplar by the size of its cluster. The false positives we saw did share syntax, but were not useful clones. For example, clones of functions registering action listeners, clones of simple constructors setting member fields, clones of equals functions using the same programming conventions, and so on. The discrepancy between the generalized and system-specific precisions is due to the aforementioned false positives being very common in this subject system, a class library. We would not expect these false positives to be as common in a different type of subject system. This is the usefulness of our cluster-based technique, it allows us to measure a generalized precision by removing the bias of the distribution of the clone varieties in the particular subject system.

If we only require the clone pairs to share syntax to be true positives, we measure a precision of 100% – indicating no false positives caused by bugs in NiCad's algorithms. Noting that some of the false positives were clones of common Java programming and API idioms, the precision of NiCad could be improved by targeting these programming patterns and filtering the detected clones. Our clustering technique could be used as a manual filtering technique, allowing the user to discard the clusters that contain false positives after a minimum inspection. In our experience with this case study we noted a few additional advantages of our technique. If we were unsure how to judge a particular clone pair we could check other exemplars of the same clone variety to make our decision. By keeping the number of clones needing valuation small, and by reducing the chance of seeing two exemplars of the same clone variety (Figure 13), we were more confident we were judging the clones consistently.

8. Related Work

Although equally as important as recall, there has been a marked lack in the measurement of clone detection precision, possibly due to the challenges involved (e.g., extensive manual efforts [17, 32], reliability of judges [13], etc). Algorithm and tool publications occasionally include a measure of precision [33, 14, 34, 35, 15, 36]. Usually the tool author checks on the order of 100 clones detected in a single or a couple software systems. Roy and Cordy [32] measure a semi-automated precision, where an automatic clone validator attempts to validate the clones, while leaving the clones it is unsure of up to manual inspection. However, their approach is limited to artificial clones created by a mutation analysis procedure and injected into the subject system.

There have been a few studies that compare the precision of multiple tools. Burd and Bailey [37] measured the precision of 5 tools by manually validating all the clones they found in a very small subject system, 1741 clones overall. Falke

et al. [16] measured the precision of 4 tools using 2 subject systems, and report requiring 11 working days to validate 9,415 clones. The most extensive study was done by Bellon et al. [17] who measured the precision (and recall) of 6 tools by manually validating 2% of the clones they detected in 8 subject systems, with the clones selected randomly. In total, 77 hours of effort were required to validate 6528 clones.

While Bellon's validated clones are publicly available [38], and can be used to measure the recall of clone detectors [17, 8], the benchmark can only be used to measure a lower bound on precision for tools that were not used to build the benchmark. This involves accepting as true positives the detected clone pairs that match those in the benchmark. This is not very effective due to the low chance of matching clones between the benchmark and modern clone detectors [8]. Also, there is concern regarding the quality of Bellon's clone corpus [11, 12, 8]. Adding a new tool the benchmark, and measuring its precision, requires extensive manual validation efforts, and is not comparable to the original tools due to undocumented clone validation procedures [16].

The most similar work to ours is by Yang et al. [39], who use supervised learning to classify clones similar to manually labeled clones in a training set with a 70-90% accuracy depending on the user, subject system and number of labels. The training set was built by manual labeling of a random selection of clones. In contrast, our work uses unsupervised learning to cluster similar clones to ensure a wide variety of clones are considered when measuring precision. Our technique efficiently measures an accurate precision using a variety of clones while minimizing biases inherit to random sampling approaches.

9. Threats to Validity

A different configuration of the experiment might provide better results. For example, a different number of clusters, or a different number of principal components to keep. We chose these parameters based on cluster quality metrics (silhouette), but these metrics do not understand the semantic goal of our clustering. This requires manually investigating the chosen clustering solution, which is too time-intensive to repeat for many permutations of the configurations. However, we have shown that the metrics do lead us to a clustering solution with good results for our motivating use-cases.

As part of our evaluation, we manually inspected clone clusters to judge if they contained a single or multiple varieties of clones. We also manually clustered randomly selected clones to judge the total number of clone varieties seen. These judgments are somewhat subjective, and are based on our experience in clone research and clone detection benchmarking. The judgments may differ if made by different researchers, although they require a clone expert to make good judgments. We documented our clone variety judgments, and ensured that we applied these judgments uniformly across the experiments. Therefore we minimize the affect this subjectivity

has on the comparison between the traditional and our cluster-based approach to measuring precision.

10. Conclusion

In this paper we presented a novel approach for efficiently measuring an accurate and generalized clone detection precision. Our approach selects clone pairs for validation from a clustering solution which guarantees a wide variety of clone pairs are considered with minimal bias. We cluster the clones using a custom vectorization process and the k-means algorithm after dimensionality reduction with principal component analysis. We evaluated our approach with NiCad as our subject clone detector and a selection of the Java class library as our subject system. We compared our approach against the traditional measurement of precision by random sampling. We found that our technique considers twice the variety of clones for the same sample size, while random sampling requires a sample size nearly three times larger to match our variety of clones considered. Traditional random sampling suffers from biases that harm the generality of the precision measured due to the uneven distribution of different clone varieties within the specific subject system, which may be different in another arbitrary software system. Our technique reduces these biases by up to an order of magnitude, and has very little bias overall. As future work, we plan to use this approach with the clones we detected in our recall experiments [28, 8] to measure the precision of the modern clone detection tools.

References

[1] C. K. Roy and J. R. Cordy, "A survey on software clone detection research," Tech. Rep. TR 2007-541, School of Computing, Queens University, 2007. 115 pp.

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

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

[4] S. Cesare, Y. Xiang, and J. Zhang, "Clonewise - detecting package-level clones using machine learning," in *Security and Privacy in Communication Networks*, vol. 127 of *Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering*, pp. 197–215, Springer International Publishing, 2013.

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

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

[7] J. Svajlenko, J. F. Islam, I. Keivanloo, C. K. Roy, and M. M. Mia, "Towards a big data curated benchmark of inter-project code clones," in *Proceedings of the 2014*

IEEE International Conference on Software Maintenance and Evolution, ICSME '14, (Washington, DC, USA), pp. 476–480, IEEE Computer Society, 2014.

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

[9] J. Svajlenko, C. K. Roy, and J. R. Cordy, "A mutation analysis based benchmarking framework for clone detectors," in *Proceedings of the 7th International Workshop on Software Clones*, IWSC '13, pp. 8–9, 2013.

[10] C. K. Roy, M. F. Zibran, and R. Koschke, "The vision of software clone management: Past, present, and future (keynote paper)," in *2014 Software Evolution Week - CSMR-WCRE'14*, pp. 18–33, 2014.

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

[12] A. Charpentier, J.-R. Falleri, D. Lo, and L. Réveillère, "An empirical assessment of bellon's clone benchmark," in *Proceedings of the 19th International Conference on Evaluation and Assessment in Software Engineering*, EASE '15, (New York, NY, USA), pp. 20:1–20:10, ACM, 2015.

[13] A. Walenstein, N. Jyoti, J. Li, Y. Yang, and A. Lakhotia, "Problems creating task-relevant clone detection reference data," in *Reverse Engineering, 2003. WCRE 2003. Proceedings. 10th Working Conference on*, pp. 285–294, Nov 2003.

[14] L. Jiang, G. Misherghi, Z. Su, and S. Glondu, "Deckard: Scalable and accurate tree-based detection of code clones," in *Software Engineering, 2007. ICSE 2007. 29th International Conference on*, pp. 96–105, May 2007.

[15] 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, pp. 176–192, March 2006.

[16] 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.

[17] 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, pp. 577–591, Sept 2007.

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

[19] Oracle Corporation, "Java SE development kit 8." https://www.oracle.com/java/index.html, 1995–2015.

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

[21] D. Arthur and S. Vassilvitskii, "K-means++: The advantages of careful seeding," in *Proceedings of the Eighteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '07, pp. 1027–1035, 2007.

[22] A. P. Dempster, N. M. Laird, and D. B. Rubin, "Maximum likelihood from incomplete data via the em algorithm," *JOURNAL OF THE ROYAL STATISTICAL SOCIETY, SERIES B*, vol. 39, no. 1, pp. 1–38, 1977.

[23] 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.

[24] MathWorks, "Silhouette plot - matlab silhouette." http://www.mathworks.com/help/stats/silhouette.html?refresh=true.

[25] DataScienceLab, "Finding the k in k-means cluster, the data science lab." https://datasciencelab.wordpress.com/2013/12/27/finding-the-k-in-k-means-clustering/.

[26] M. Steinbach, G. Karypis, and V. Kumar, "A comparison of document clustering techniques," in *In KDD Workshop on Text Mining*, 2000.
[27] K. P. Murphy, *Machine Learning: A Probabilistic Perspective*. The MIT Press, 3 ed., 2012.
[28] J. Svajlenko and C. Roy, "Evaluating clone detection tools with bigclonebench," in *Software Maintenance and Evolution (ICSME), 2015 IEEE International Conference on*, pp. 131–140, Sept 2015.
[29] J. Cordy, "The txl programming language." http://www.txl.ca/.
[30] J. Ullman, "Clustering." http://infolab.stanford.edu/ ullman/mmds/ch7.pdf.
[31] MATLAB, *(R2015b)*. Natick, Massachusetts: The MathWorks Inc., 2015.
[32] C. K. Roy and J. R. Cordy, "A mutation/injection-based automatic framework for evaluating code clone detection tools," in *Software Testing, Verification and Validation Workshops, 2009. ICSTW '09. International Conference on*, pp. 157–166, April 2009.
[33] M. Gabel, L. Jiang, and Z. Su, "Scalable detection of semantic clones," in *Proceedings of the 30th International Conference on Software Engineering*, ICSE '08, (New York, NY, USA), pp. 321–330, ACM, 2008.
[34] R. Komondoor and S. Horwitz, "Using slicing to identify duplication in source code," in *Proceedings of the 8th International Symposium on Static Analysis*, SAS '01, (London, UK, UK), pp. 40–56, Springer-Verlag, 2001.
[35] J. Krinke, "Identifying similar code with program dependence graphs," in *Reverse Engineering, 2001. Proceedings. Eighth Working Conference on*, pp. 301–309, 2001.
[36] C. Roy and J. Cordy, "Nicad: Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization," in *Program Comprehension, 2008. ICPC 2008. The 16th IEEE International Conference on*, pp. 172–181, June 2008.
[37] E. Burd and J. Bailey, "Evaluating clone detection tools for use during preventative maintenance," in *Source Code Analysis and Manipulation, 2002. Proceedings. Second IEEE International Workshop on*, pp. 36–43, 2002.
[38] S. Bellon, "Detection of software clones." http://www.bauhaus-stuttgart.de/clones/. Accessed: 2015-01-15.
[39] 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.