On the Use of Machine Learning Techniques Towards the Design of Cloud Based Automatic Code Clone Validation Tools

Golam Mostaeen, Jeffrey Svajlenko, Banani Roy, Chanchal K. Roy, Kevin Schneider
Department of Computer Science,
University of Saskatchewan, Saskatoon, Canada
Email: {golam.mostaeen, jeff.svajlenko, banani.roy, chanchal.roy, kevin.schneider}@usask.ca

***Abstract*—A code clone is a pair of code fragments, within or between software systems that are similar. Since code clones often negatively impact the maintainability of a software system, a great many numbers of code clone detection techniques and tools have been proposed and studied over the last decade. To detect all possible similar source code patterns in general, the clone detection tools work on syntax level (such as texts, tokens, AST and so on) while lacking user-specific preferences. This often means the reported clones must be manually validated prior to any analysis in order to filter out the true positive clones from task or user-specific considerations. This manual clone validation effort is very time-consuming and often error-prone, in particular for large-scale clone detection. In this paper, we propose a machine learning based approach for automating the validation process. In an experiment with clones detected by several clone detectors in several different software systems, we found our approach has an accuracy of up to 87.4% when compared against the manual validation by multiple expert judges. The proposed method shows promising results in several comparative studies with the existing related approaches for automatic code clone validation. We also present our experimental results in terms of different code clone detection tools, machine learning algorithms and open source software systems.**

***Keywords*—Code clones, validation, Machine Learning, Clone Management.**

I. Introduction

Copying and reusing certain pieces of existing code directly or with alteration into another location is a common programming practice in a software development life cycle [1]. Researchers agree upon four primary clone types [1] : Type-1 clones are syntactically identical code fragments, regardless of the presentation style, comments, and white spaces. Type-2 clones are copy and pasted code where identifier names and types have been changed. Type-3 clones are modified copies of the original code with statement-level changes (e.g., added, modified or removed). Syntactically dissimilar code fragments that implement the same or similar functionality are termed as Type-4 clones [1]. Some of the recent research shows that on average around 7% to 23% of total codes of a software system are duplicated or cloned from one location to another [2], [3], [4]. Although code cloning is often done intentionally to accelerate the development process and also not all code clones are harmful [5], the existence of some of them can inflate software maintenance costs as clones are one of the major causes of creation and propagation of software bugs throughout the system [1], [6].

At least 198 code clone detection tools and techniques have been proposed and developed to automate the clone detection process until 2017 [7], as a result of extensive research in this specific area over the last decade [1], [8], [9], [10], [11], [12], [13]. These tools return a list of possible code clone pairs available in a given software system. Except for Type 1, the other types of code clones (e.g., Type 2, 3 and 4) undergo different changes over time and gets complicated to be detected by a simple string matching algorithm. For example the identifiers or functions names may be changed, some code statements may be added, modified or removed, a portion of the code clones might undergo several other syntactical changes or even the complete implementations might be changed for the same functionalities in any other locations and so on [11]. All these modifications over time make the searching problem much more complicated [1].

In order to handle those complex source code structures while still detecting all possible code clone pairs, the clone detection tools undergo a lot of generalization of the original source codes like pretty-printing [11], [1], normalization of the identifiers [8], [11], forming syntax tree [14] of the code fragments and so on, and often return pairs of code fragments those are not essentially clones. These are pairs of code fragments that are only coincidentally similar, or are otherwise considered not a valid clone by users [15]. Besides, some research shows that the definition of true positive code clones especially in case of Type 3 and Type 4 clones are subjective and might also be different for different users, domains, programming languages or software systems [16], [17], [18].

For these case-specific considerations, users often need to manually validate if the result is a true clone or not before using this information for the given specific scenarios such as source code refactoring or other software maintenance tasks at hand [19], [18]. This manual validation process even becomes a hindering factor as the software system evolves in size over time [15] (e.g., research shows that JDK 1.4.2 contains 204 KLOC reported code clone which is 8% of the total total lines of code [15], [19]. 15% of the total lines of code of the Linux kernel has been reported as code clone which is 122 KLOC [20]). Recent studies focus and show that the usability of the code clone detection tools can be improved significantly by considering the user or requirement specific patterns in addition to the general syntax level matching [21], [18], [22].

In this paper, we propose a machine learning based approach for predicting the user code clone validation patterns. The proposed method works on top of any code clone detection

tools for classifying the reported clones as per user preferences. The automatic validation process for a user, thus can accelerate the overall process of code clone management and helps faster acquiring of required information out of the clones in comparison to time consuming manual validation process. We also extend the proposed method with a cloud based architecture to ensure compatibility of the proposed method with any of the existing code clone detection tools.

A cloud based prototype system has been developed as a proof of concept of the proposed method (available for public use [23]), which shows an accuracy of up to 87.4% when compared against the manually validated code clones by multiple expert judges. In this paper, we also present our experiments and evaluation results in multiple use-cases such as, comparison to related existing methods, multiple code clone detection tools, different open source projects, multiple machine learning algorithms and so on. From our investigations, we got promising results in comparison to existing methods for the code clone validation problem with different experimental setups.

Our work makes three main contributions. First, we studied the data distribution for the clone classification problem with several extracted features. Our findings on these feature sets and data distribution analysis can help better understand the clone classification problem and thus adds the possibility of further result improvement in this research area. Second, we conducted detail comparative study with 12 different machine learning algorithms for the clone classification. To the best of our knowledge, no previous studies were done that focused on a comparative study of different machine learning algorithms for clone classification problem. Our observations on strengths and weaknesses of several machine learning algorithms on clone classification can contribute to the future research on this area for further improvement of the learning model. Finally, we present a cloud-based prototype system [23] which is compatible with any existing clone detection tools as per the concepts and framework of the proposed method.

II. Proposed Approach

The proposed method uses machine learning models for predicting the user-specific code clone validation. The models are first trained based on manually validated code clone sets from the corresponding users. The trained models are then used for improving the reported code clones from clone detection tools, by predicting the user specific validation patterns. We further extend the proposed method by providing a cloud based architecture for taking advantages of high performance computation and compatibility with any of the existing code clone detection tools.

A. Clone Dataset: Software Systems

Detected clones from any subject software system can be used for training the machine learning model. For our experiments, we used clones from IJaDataset 2.0 [24] - a large inter-project dataset of open-source Java systems. To test the generality of the proposed method, 5 different publicly available and state-of-the-art tools namely NiCad [25], Deckard [12], iClones [26], CCFinderX [8] and SourcererCC [27] were used to detect clones separately out of the benchmark. We have chosen to work on this dataset because a good number of recent

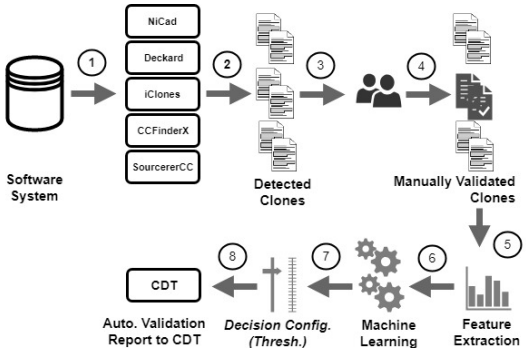


Fig. 1: Workflow of the proposed approach.

research works on code clones has been carried out on these open source projects [27], [28], [29] and thus we can have a common ground for evaluating the proposed approach.

B. Manual Code Clone Validation for Training

A subset of the reported code clones from the clone detection tools are then provided to the user for manual validation (e.g., Step 3, Figure 1). The corresponding user validation results are stored in a database which is later used for training the machine learning model. Reported clones from clone detection tools are used to create clone database, K. Clones from K, are manually marked as true or false positive by the user. Reported code clones are thus grouped into two disjoint sets K_t and K_f - representing true positive and false positive clone groups respectively such that, $K = K_t \cup K_f$ and $K_f \cap K_t = \emptyset$. K_t and K_f are used for training the machine learning models. For our experiments, randomly 400 clone pairs were selected from the used dataset (i.e., Section II-A) and manually validated from each of the five clone detection tools separately to mitigate any possible biases.

C. Feature Extraction

To target automatic validation even beyond Type 2 clones while addressing the limitations of the existing methods (as discussed in Section VII), we focused on extracting more informative features in addition to simple token sequence matching.

```
try {
    if (args.length == 0) {
  throw new Exception(
      "The first argument must be the
        class name of a kernel");
    }
    String associator = args[0];
    args[0] = ">";
    System.out.println(evaluate(associator
        , args));
}
```

Listing 1: Fragment 1.

```
try {
    if (args.length == 0) {
 throw new Exception(
 "The first argument must be the name
       of a "
 + "clusterer");
    }
    args[0] = "?";
    Clusterer newClusterer =
       AbstractClusterer.forName(
       ClustererString, null); //object
        from abstract clusterer
    System.out.println(
       evaluateClusterer(newClusterer,
       args));
}
```

Listing 2: Fragment 2.

For feature extraction, we tried to mimic the human validation patterns as obtained from our investigation and related works [18]. For example, Listing 1 and Listing 2 show the code fragments of one of the detected clones from Weka [30] software system, that needs to be validated. Although the code clone fragments exhibit enough similarities to be detected by code clone detection tools, the human validation

preferences may be impacted by different factors and vary from user to user [18]. For example, line 1-3 of the code fragments exhibit Type 1 clone, line 4 exhibit Type 2 clone, while line 5 and beyond shows the existence of Type 3 code clones. Code similarity for these multiple clone types might significantly differ in influencing the user-specific validation decision. While the code clone detection tools in general work via simple syntax or structure level matching [1], [8], [12], [13] without any such preference considerations, the related existing token sequence based clone validation approaches targeted Type 2 clones mainly [18]. With an attempt to validating beyond Type 2 clones, feature extractions were considered by three levels of code clone normalization independently. For example, code fragments similarity followed by only removing any comments (e.g., in line 8 of Listing 2) and pretty printing

```
1  try {
2      if (X.X == 0) {
3    throw new X(
4        "string");
5      }
6      X X = X[0];
7      X[0] = "string";
8      X.X.X(X(X, X));
9  }
10
```

Listing 3: Trans. frg. 1.

```
1  try {
2      if (X.X == 0) {
3   throw new X(
4   "string"
5   + "string");
6      }
7      X[0] = "string";
8      X X = X.X(X, null);
9      X.X.X(X(X, X));
10 }
```

Listing 4: Trans. frg. 2.

denotes the degree of Type 1 relation among the subject clone pairs. Similarly, Listing 3 and Listing 4 show the transformed clone fragments from Listing 1 and Listing 2 respectively, after first blind renaming of identifiers and then applying consistent normalization of literals [31]. These transformations allow the corresponding modifications of literals and identifiers and thus provide similarity feature information for the targeted clone pairs. We used TXL [31] for the required pre-processing and source transformations. The corresponding numerical similarity value between the two code clone fragments f_1 and f_2 is then calculated as, $\xi(f_1, f_2) = 1 - max(C(\mathcal{O}_d)/|f_1|, C(\mathcal{O}_i)/|f_2|)$, where $C(\mathcal{O})$ and $|f|$, represent the counts of minimal changes required to transform from one clone fragment to another and length of the corresponding code clone fragment respectively.

We also used several other features to get more structural information about the two code clone fragments. For example, our intuition was if the code clone fragments are significantly different in sizes, human validator may be more likely to mark them as false positive. The difference, $|\alpha - \beta|$ was considered as one possible feature, where α and β represent the sizes of code fragments. However, for a smaller code clone pair versus a larger code clone pair the considerations might be different. So the average size of the code clones $(\alpha + \beta)/2$, were also considered. That average value captures sort of the size of the clone and difference captures if the code fragments are rather mismatched in size.

Some of the popular code clone detection tools use source transformations like consistent renaming of identifiers, normalization of literals and so on, as part of their workflow for code clone detection [1]. While for a subset of our extracted feature set we use similar source transformation, our work is significantly different from those code clone detection tools in terms of targets and final feature set calculation. Code clone detection tools generally use these source transformation for code fragment comparison (for example, token sequence matching). However, we used these source transformation as an intermediate steps for a subset of feature calculation and also used for learning human validation patterns unlike string sequence matching adapted by code clone detection tools.

To the best of our knowledge no previous works on clone validation used similar feature set, hence before finalizing the feature selection for building the machine learning models, we conducted several studies on data distribution with the extracted features. Table I shows a summary of the feature set from our study on data distribution (Section III).

D. Training Machine Learning Models for Clone Classification

As we have presented the workflow of the proposed method in the above discussion, it uses supervised machine learning algorithm for learning the classification pattern of the user specific clone validation (i.e., Figure 1, Step 6). The supervised classification algorithm are trained on the manually validated datatset $K = \{(\mathbf{x}_1, \mathbf{y}_1), (\mathbf{x}_2, \mathbf{y}_2)...(\mathbf{x}_m, \mathbf{y}_m)\}$, for $\mathbf{x}_i \in \mathbb{R}^n$ and $\mathbf{y}_i \in \mathbb{R}^l$, where n and l represent the extracted clone feature set and clone validation labels respectively. The machine learning algorithm is then trained on dataset K, to learn a function f, such that f can map from $\mathbb{R}^n$ to $\mathbb{R}^l$, representing the class probability for being true or false positive for the given pairs of code clones.

We got best result using Artificial Neural Network (ANN) from our study on multiple machine learning algorithms (Section IV) for clone classification. From the training dataset K, for $\mathbf{x}_i \in \mathbb{R}^n$ in the input layer, $\mathbf{y}_i \in \mathbb{R}^l$ in the output layer and one hidden layer with k nodes, ANN learns the function: $f(\mathbf{x}) = \sigma(W_{ho}^{\intercal} \cdot \sigma(W_{ih}^{\intercal} \cdot \mathbf{x} + \theta_h) + \theta_0)$ where, $W_{ih} \in \mathbb{R}^{n \times k}$ and $W_{ho} \in \mathbb{R}^{k \times l}$ denotes the connection weights from the input layer to the hidden layer and hidden to output layer respectively. θ and σ denotes the layer bias and neuron activation function respectively. For the training phase the Neural Network was run with different values of k (to investigate the optimal network configuration), for a number of epochs until it converges with a maximum limit of 1000 epochs. Softmax activation function was used for the output layer. The model was trained and tested using 10-fold cross validation. The Neural Network converged within a range of 500 to 600 epochs for $k = 107$, giving an accuracy of 87.4%.

E. Configuring the Prediction Decision

The machine learning models classify the test code clone pairs using the extracted feature vector, $\mathbf{x}_t$. Probabilistic classifiers learns a function f, such that $f(\mathbf{x_t})$, assigns probability

TABLE I: Features Based on Distribution Analysis.

Feature	**Distribution Mean Difference, $\Delta\mu$**
Line Sim. (Type-1 Norm.)	0.3998
Line Sim. (Type-2 Norm.)	0.3701
Line Sim. (Type-3 Norm.)	0.3602
Token Sim. (Type-2 Norm.)	0.3447
Token Sim. (Type-1 Norm.)	0.3105
Token Sim. (Type-3 Norm.)	0.2537
Function Intersected	0.2364
Unmatched Braces	0.2078

values, $\hat{\mathbf{y}}_t$ for the two classes, where $Pr[\mathbf{y}_t = (1, 0)]$, represents the probability of belonging to true positive clone class. A decision threshold $\gamma[0, 1]$ is used to tune the validation output quality (i.e., Step 8, Figure 1). A test clone pair is reported as true positive if $Pr[\mathbf{y}_t = (1, 0)] \geq \gamma$. The default value of γ is set to 0.5 for deciding the clone validation (i.e., classified as true positive code clone if $Pr[\mathbf{y}_t = (1, 0)] \geq Pr[\mathbf{y}_t = (0, 1)]$). So, on setting this γ value towards its upper limit (i.e., 1.0), the validation becomes more strict for classifying clones and will return only those clone pairs having higher probability of being true positive clones. Similarly, one can decrease the value of γ to make the proposed method more tolerant for classifying the clones in true positive class.

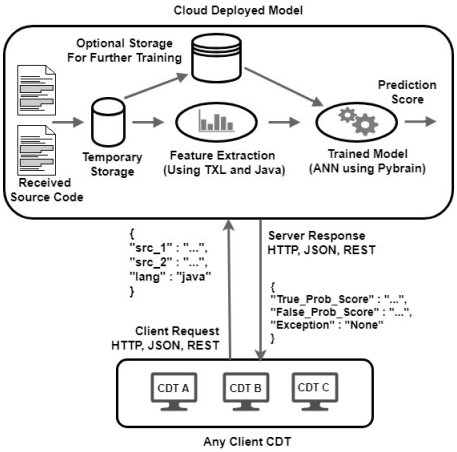
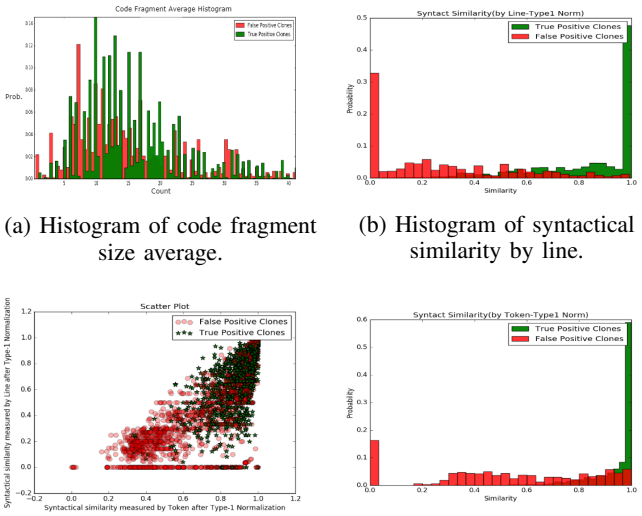


Fig. 2: Cloud model for compatibility with existing code clone detection tools.

F. Cloud Architecture for Clone Classification

In addition to using the trained model locally for code clone classification, we also extend the proposed method with a cloud based architecture for several additional advantages. Figure 2 shows an overview of the architecture. Reported code clones from a target code clone detection tool are sent to the server for validation using HTTP request. The request mainly contains the targeted code clone pairs that need to be validated. The request can optionally contain some additional information to be used by the proposed method. For example, the classification models to use, possible configuration for the classification model, language of the code clone source code and so on. The communication with the server is done using JavaScript Object Notation [32] (i.e., in key-value pairs as an example shown in Figure 2). On receiving the subject clones to validate, the required features are extracted to build feature vector $\mathbf{x}_t$, which is then used by the trained machine learning model to get the probability score $f(\mathbf{x}_t)$. The corresponding scores are then sent back to the clone detection tool for displaying validated result in the user end. There are several opportunities and advantages of the cloud deployment, such as, *compatibility with any existing clone detection tools (i.e., via RESTful web services)* , *improvement in the training phase*, *opportunities for higher processing power* and so on.



(a) Histogram of code fragment size average.

(b) Histogram of syntactical similarity by line.

(c) Syntactical sim. by line vs token.

(d) Histogram of syntactical similarity by token.

Fig. 3: Classification result analysis for different types of code clones

III. Study on the Data Distribution

Machine learning algorithms try to recognize any available underlying patterns using feature set from a given dataset. Hence, feature distribution study is important and helps to better understand the classification problem [33]. Here we present our experimental studies on the underlying data distribution analysis in terms of extracted features of the proposed approach (Table I), which is also a contribution of our work for better understanding the clone classification problem and future improvement in the research domain.

We analyzed the extracted feature of the code clone pairs for both true positive and false positive manually validated clones in an attempt to find its contribution score for clone classification. Figure 3a shows the distribution of the average code clone fragment feature for the true positive and false positive clone classes. From the figure, we see that the average code fragment size shows much randomness, both for true positive and false positive clones. The distribution of this feature almost overlaps with one another for the two classes: true positive and false positive code clones. This overlapping pattern suggests that this feature represents a minimal information about the human validation pattern in two classes and thus yields a very low possible contribution score for training the machine learning algorithm for validation.

Besides for extracting some other features, we normalized the code clone pairs by 3 levels, namely: Type 1, Type 2 and Type 3. Then for each level of normalizations, syntactical similarity was measured both by lines and by tokens, for the clone pairs resulting 6 different possible features (as discussed in Section II-C). To visualize any underlying distribution of the features their normalized histogram were plotted both for true positive and false positive clones. Figure 3b shows one of such plottings that is based on the similarity measured by lines after Type 1 code normalization. It is noticeable from the figure that the distribution of the feature is comparatively

better than the average code fragment line feature in terms of validation. Although the distribution of true positive and false positive clones are not completely linearly separable with this feature, still the two classes are somewhat distinguishable. The distribution indicates a possible better contribution score for the human validation prediction than the average clone fragment sizes. Figure 3d also shows somewhat similar results in case of syntactic similarity measured by tokens after Type 1 Normalization of the code fragments.

We also carried out several studies to find out any underlying relationships between different features for possible clustering of the two clone classes. For example, we tried to figure out if there are any underlying relationships available for different types of similarity measures that can give any potential information about the clustering of the two clone classes. We plotted our several study result for visualization in an attempt to notice any distinguishable separation or clusters. Figure 3c is one of such study results that shows the scatter plot on syntactical similarity measured by line versus tokens after Type 1 Normalization of the code clone pairs. While the investigation results suggest absence of any distinguishable cluster, the feature distribution mean difference, $\Delta\mu$ (Table I) shows some positive results and hence promote usage of classification algorithms for the clone classification problem.

IV. Machine Learning Algorithms: A Comparative Study

Although we got the best result for classification with ANN (hence used it in our proposed method, Section II-D) and conducted several experiments (Section V) with proposed method, here we present a comparative study of 12 differnt machine learning algorithms.

A. Bayes Classifiers

From the extracted code clone feature vector $\mathbf{x} = (x_1, x_2, \cdots, x_n)$, we experimented with Naive Bayes Classifier - a conditional probability model [34], for classification of the clonesets into two clone classes - C_T and C_F, representing true and false positive validated clone classes respectively. We used kernel density estimation [34] for the likelihood calculation as most of the selected feature values are continuous numerical values. With the described configurations the classifier showed an accuracy of 83%, with 0.831 and 0.830 of precision and recall respectively. However, it can often be a strong assumption by the classifier to consider independence among the extracted features (i.e., as Naive Bayes) for a given class C, for the clone classification problem. Because, by the definition of the code clones [1], it is usual that part of a Type 2 clone can contain Type 1 clone. Similarly, Type 3 clones can also contain fractions of Type 2 or Type 1 clones. So by induction, it is expected that the extracted similarity features for a given clone class have some sort of correlation among them [18]. Figure 4 illustrates the correlation among some of the extracted features. As it is noticeable from the figure, the clone structural features such as average line, line differences and so on show relatively lower correlation with other features. However, the extracted similarity based features among clone pairs after different levels of normalization show significantly higher correlation among them. From these findings, we also experimented with Bayesian Network Classifier [35] - that considers and learns

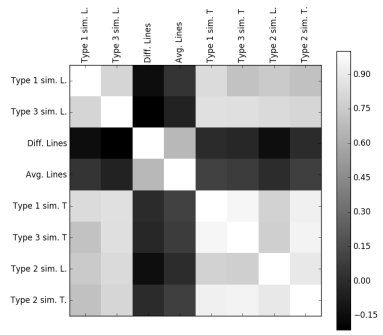


Fig. 4: Correlation among feature subset.

the possible dependency relations among the features. Unsupervised learning was used (e.g., via Minimum Description Length (MDL) [36] scoring method) to build the dependency network among the features.

Although Bayesian Network tries to mitigate the strong assumption made by the Naive Bayes, we found that the two classifiers perform relatively the same for the clone classification problem with the used features. In fact, in some cases, Naive Bayes outperformed the Bayesian Classifier (e.g., as illustrated in Figure 5). The behaviour is not totally unexpected though, as Friedman *et al.* [35] showed a detailed study on this. Errors while learning the dependency network from the training set was presented as possible reasoning for such behaviour.

B. Decision Tree Classifiers

For predicting the target variables, these classifiers build decision tree from the input variables of the used feature vector, $\mathbf{x}$ [37]. The internal nodes of the tree correspond to different input variables. The values of the corresponding input variables define the edges connecting nodes and each leaf denotes different target variable for the classification. Pruned C4.5 decision tree [38] showed an accuracy of 84%. The obtained precision and recall are 0.849 and 0.848 respectively. Random Tree - a variant of the classifier considers K random input variables at steps for generating the decision tree [37]. The obtained accuracy for clone validation was 79%.

TABLE II: List of Operations used to Create Artificial Code Clones via Mutation Framework [39].

Clone Types	Modification Operations
Type-1	Addition/Removal of white-space
	Changing the code comments
	Addition/Removal of newlines
Type-2	Systematic renaming of identifiers
	Arbitrary renaming of identifiers
	Change in value of literals
Type-3	Insertion/Deletion within lines
	Insertion/Deletion of lines
	Modification of whole lines

TABLE III: Result on Artificial Code Clones.

Accuracy	Precision	Recall	F1-Score
90%	0.89	0.99	0.93

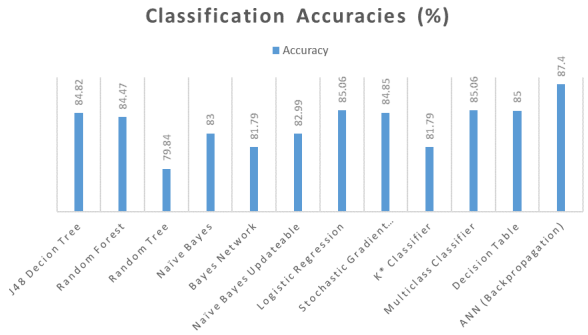


Fig. 5: Exploring the classification performance of multiple machine learning algorithms.

V. Experiments and Evaluations

A. Implementation Details

We developed a cloud based framework as per the proposed method for machine learning based automatic clone validation and also experimented with its performance in different experimental setups [23] [1]. For collecting the user-specific training data, a cloud based web application was first developed. We used Python 2.7, as the server side language. The web application was developed using Flask [40] - a microframework for Python. The system server can be populated by code clones reported by different code clone detection tools for user-specific validation. The system iteratively displays the code clones to the users for manual validation. We used CouchDB [41] - a NoSQL database system, that supports easier scaling up and distributed computing for Big Data. We selected CouchDB to take advantage of this feature of the database for handling large amount of code clones in our future works.

B. Experimental Evaluation on Artificial Clones

Evaluation of code clone related tools and techniques can often be critical as the validation of some of the types of code clones as true or false positive vary significantly from person to person [16], [17]. So to get more concrete information about the validation accuracy by the trained model, we were interested of evaluating the system with artificially generated clones before testing on real clones from different software systems (e.g., as presented Section V-C). We used Mutation Framework [39] for creating such artificial code clones. The framework takes a code fragment as input, performs mutation by random edit operations on the code fragment and inject the resultant fragment to artificially create a clone pair.

We used nine different mutation edit operations on the source codes as listed in Table II. These operations create three different types (Type-1, Type-2 and Type-3) of true positive clones which are mostly simpler, straight forward and void of any subjective biases by the individuals. We used different original code fragments from BigCloneBench [42] to create 3750 such artificial true positive code clone pairs. Our target was to test the performance of the proposed method on validating those artificial true positive clones, so along with them we mixed 840 randomly selected false clones from dataset. We then applied the proposed method on the clones for validation. We got comparatively better accuracy on these artificially created clones as shown in Table III. The possible reasoning for this relatively higher performance is that, while the artificially created clones are void of subjective biases, they are often very similar to one another and comparatively easily distinguishable.

[1]Cloud based prototype system: http://p2irc-cloud.usask.ca/ccv, GitHub: https://github.com/pseudoPixels/ML_CloneValidationFramework

TABLE IV: Used Clone Detection Tools for the Experiment.

CDT	Ver.	Tool Configuration
iClones [26]	0.2	mintokens = 50, minblock = 20
NiCad [11]	4.0	blocks, 30%, 6-2500 lines, blind-renaming, abstract-literal
SimCad [43]	2.2	generous, 6+ lines, blocks
CloneWorks [28]	0.2	Type-3 Aggressive, 6 lines, blocks
Simian [44]	2.4	6 lines, ignore overlapping blocks, balances parentheses
Ctcompare [45]	3.2	50 tokens, 3 replacements

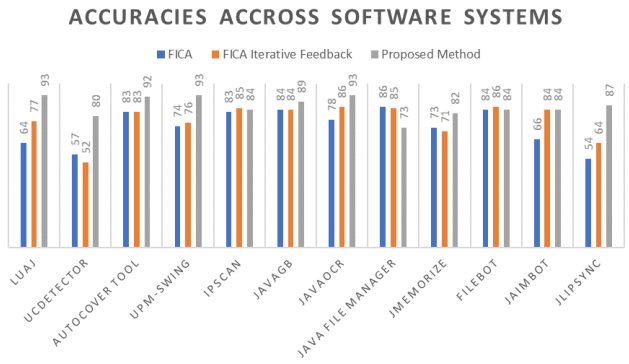


Fig. 6: Comparison of the proposed method with related existing methods (across different software systems).

C. Evaluation on Different Software Systems: A Comparative Study with Existing Methods

The proposed method shows a promising result with an accuracy of 87.4% via 10-fold cross validation on the data set as discussed in Section IV. The result also exhibits confidence as the used dataset is comparatively larger and contains a number of diverse software projects. However, we were also interested to see how the proposed method works for different software projects. We selected 12 completely different open source projects that were not used in any of the previous training or testing phases. We used different code clone detection tools (Table IV) for detecting the code clones available in those open source software projects. The reported code clones from code clone detection tools were then manually validated by different users. Besides, to compare the performance of the proposed method with similar existing method - FICA [18],

TABLE V: Comparison with Existing Systems.

Software System	LoC clones [1]	Avg. Lines [2]	Avg. Tokens [2]	FICA		FICA Iterative		Proposed Method	
				Precision	Recall	Precision	Recall	Precision	Recall
Luaj	36155	15	79	0.969642857	0.629930394	0.97619047	0.769230769	0.979827089	0.945319741
Ucdetector	4388	11	67	0.951219512	0.549295775	0.971428571	0.478873239	0.895833333	0.883561644
Autocover Tool	3989	13	50	0.830188679	0.956521739	0.843137255	0.934782609	0.926315789	0.967032967
Upm-swing	13243	11	73	0.989690722	0.738461538	0.994923858	0.753846154	0.985971944	0.944337812
ipscan	7082	10	58	0.863247863	0.918181818	0.922330097	0.863636364	0.964912281	0.800970874
JavaGB	24211	9	58	0.784722222	0.875968992	0.792114695	0.856589147	0.9	0.861878453
JavaOcr	7699	18	90	0.970588235	0.76744186	0.973684211	0.860465116	0.988304094	0.933701657
JavaFileManager	25898	12	68	0.962962963	0.882352941	0.967254408	0.868778281	0.941807044	0.725235849
jMemorize	13109	10	44	0.926829268	0.619565217	0.933774834	0.658878505	0.91576087	0.828009828
FileBot	18369	11	59	0.765217391	0.946236559	0.791855204	0.940860215	0.969581749	0.676392573
JAIMBot	14096	12	83	0.993710692	0.619607843	0.98156682	0.835294118	0.987980769	0.825301205
JLipSync	3671	28	158	1	0.517241379	1	0.620689655	1	0.857142857

[1] Some of results are combination of detected clones from multiple clone detection tools (as listed in Table IV)
[2] Average per code clone fragment

we contacted and got the source code[2] from the corresponding authors.

The trained model was used for predicting the user clone validation for each of the projects. Figure 6 illustrates the comparative accuracies for the existing and proposed approaches for different software systems. As noticeable from the graph the proposed approach showed better accuracies for most of the systems. Besides, to test the result quality, system-wise

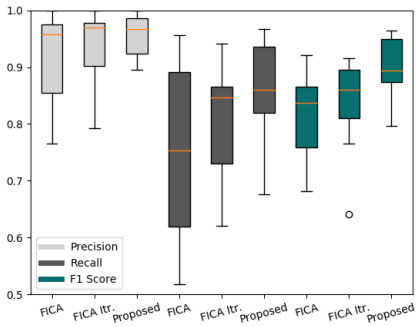


Fig. 7: Result quality comparison of the methods.

precision and recall were calculated for the approaches. The obtained result has been presented in Table V. As some of the values have been highlighted in the table, it is noticeable that for most of the cases the precision and recall values of existing methods are lower in comparison to the proposed approach. The result is also noticeable in the box plot in Figure 7. The box plot illustrates that the mean Precision, Recall or F_1-Score for the existing approaches are relatively lower than the proposed. Besides, the plot also depicts a higher variation in the result qualities for the existing approaches. In comparison, the proposed method shows promising and consistent results with lesser variation in the result qualities.

[2]Authors of FICA made the source code available for research purpose at https://github.com/farseerfc/fica

D. Evaluation with Different Code Clone Detection Tools

As the proposed method works based on the clones reported by different clone detection tools, it is also important to evaluate its performance in conjunction with different clone detection tools. We collected the clone reports from six different clone detection tools based on open source projects. Besides, as the clone detection result quality might vary greatly based on the complexity or source code structure of a given software system, to generalize we selected 500 clone pairs detected by six clone detection tools (e.g., Table IV). Users reported 315 and 185 of them as true and false clones respectively on their manual validation. We then applied the proposed method to find out its validation performance. The obtained result was: True Positive (TP) =311, False Negative (FN) =4, True Negative (TN) =115 and False Positive (FP) =70. It is noticeable from the experimental result that it could successfully validate almost all the true clones with a precision of 0.82 in comparison to the average precision of the tools (0.63). This improvement can even be more useful for large-scale software systems.

E. Evaluation of the Approach as a Clone Detection Tool Itself

While the proposed method works on top of detected possible clones as other existing state-of-the-art clone comprehension and management techniques [19], [18], [46], [47], [22], [48], we were also interested to experiment the performance of the machine learning model for detecting clones directly on any target software systems. That is instead of pairs of detected clones, extracted potential clones (i.e., smallest pieces of codes for clone consideration [11]) from a software system are passed through the learned model for the corresponding validation prediction scores.

For the evaluation, the detected clones were then independently validated by three expert judges. To mitigate any possible biases in the validation process, we adopted voting system among the judges for validation decision (i.e., a clone pair is considered as true positive iff at least two of the judges mark it as true positive independently and so on). The model showed promising results as presented in Table VI for four random software projects. The results also show impacts of different tool configurations.

To further investigate the performance of the machine learning model, judges were also asked to provide confidence scores (i.e., in the range of [0,1]) for their validation decision for the corresponding code clone pairs. The comparison of the manual confidence scores (i.e., averaged across the judges to mitigate biases) provides good insights for the comparison against the model returned probability scores for clone validation. Figure 8 illustrates the comparison for some random true positive clone pairs of two software systems. For majority of the clone pairs, the graph shows promising convergence between predicted and manual validation scores.

TABLE VI: Direct Clone Detection on Software Systems.

Software System	LoC	Min. Lines	Max. Lines	Thresh.	Precision
aTunes	1352	11	700	0.75	0.95
Java OCR	1435	10	1200	0.75	0.89
JLipSync	103	12	1000	0.75	0.95
JHotDraw	6966	5	1200	0.5	0.71

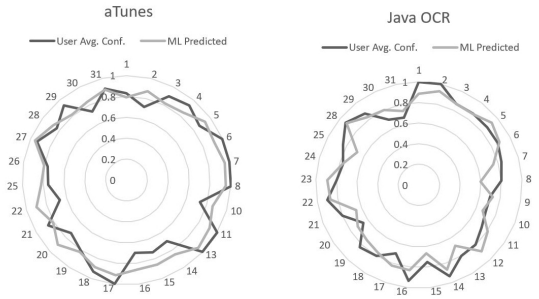


Fig. 8: Machine learning prediction vs. user validation scores.

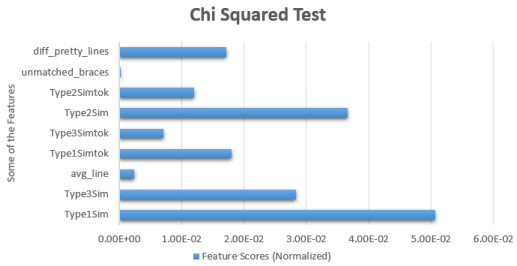


Fig. 9: Feature score via Chi-Squared test.

VI. Result Discussion

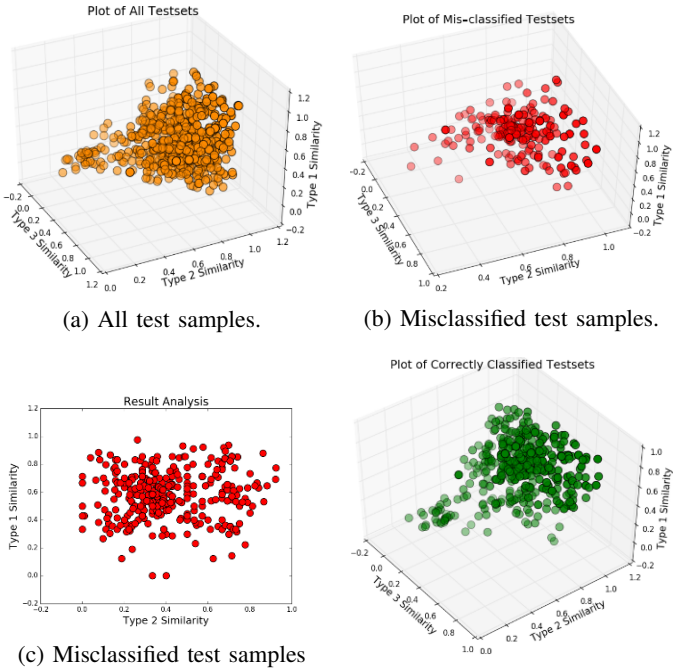
One of the major criticisms of ANNs are their being black boxes, since no satisfactory explanation of their behaviour has been offered [33]. However, assuming the ANN as black box in the middle of input sets and its predicted decision we tried to find out if there is any biases on any feature for the output decision. Based on the classified test samples by the algorithm, we calculated feature contribution scores using Chi Squared Test. If the score is too high for a particular feature in comparison to the rest, then it gives some idea about the Neural Network being biased to the particular feature. Figure 9 shows the normalized scores of some of the selected features having higher scores out of all possible extracted features. From the figure, it is noticeable that the normalized score is kind of randomly distributed over the features rather than being completely dominated by one or more features. This demonstrates that the trained model is not noticeably biased on any feature(s) on its decision making. Besides the top 3 scores are found to be Type 1, Type 2 and Type 3 code clone similarities respectively which is kind of logical for the stated clone validation problem. Chi-Squared test also supports these findings as noticeable from its low corresponding feature score in classification.

Another important aspect to analyze from the proposed method classification result is to see if it fails or succeeds only for particular type of clone(s). For example, it might be that the model can only validate Type 1 clones and cannot validate the other complex type of clones. Especially, Type 3 clone is different and difficult to validate in comparison to Type 1 or Type 2 code clones. To analyze if there are any such failure or success patterns for validation in the proposed method, we plotted the classification result in 3D space where the axes represent 3 different types of clones: Type 1, Type 2 and Type 3. The plotted results are shown in Figure 10. Figure 10a shows the scatter plot for the test samples along the 3 axes each representing 3 different types of clone similarity. From the plot we can notice the test samples are randomly scattered in the 3D space representing the presence of all types of code clone being available in the test samples. Figure 10b shows the scatter plot of the test samples that our proposed method misclassified. The randomness of the scatter plot suggests that the proposed method did not fail to classify any particular type of code clones. For example, if the algorithm would fail to correctly classify all the Type 3 clones then in the scatter plot all the misclassified test sample plot would more or less align along the Type 3 Clone Similarity axis and so on. Besides Figure 10c shows a single plane (e.g., Type 1 vs Type 2 plane) of the plotting for easier visualization. From this plot, the randomness is even clearly noticeable. From those study on the misclassified test samples by the proposed method we can get some information that it did not fail for any particular type of clone. Similarly, the proposed method can successfully classify all the three types of code clones as we can notice from the randomness of the correctly classified test samples in Figure 10d.

VII. Related Works

Although several methods and techniques have been proposed over the last years for maintenance, organization or classification of code clones, a very few of them focused on aiding the huge manual user specific validation task of the reported code clones. Yang et al. studied the similar problem for user specific code clone classification in their work - FICA [18]. The user specific clone classification in FICA is done by token sequence similarity analysis using '*Term-Frequency - Inverse Document Frequency*'- (TF-IDF) vector. As FICA learns user specific validation completely based on token sequence the validation accuracy gets significantly lower as the target clone goes beyond Type 2 as also noticeable from their study.

Tairas et al. [19] used Latent Semantic Indexing (LSI) on the identifiers of detected clone fragments by code clone detection tools to classify the code clones based on their functionality or semantic behaviours. Marcus et al. [46] and



(a) All test samples.

(b) Misclassified test samples.

(c) Misclassified test samples (2D view).

(d) Correctly classified test samples.

Fig. 10: Classification result analysis for different types of code clones.

Kuhn et al. [47] worked on a similar problem of clone classification based on the semantic topics of the detected code clone fragments. Higo et al. [22] proposed a metric-based approach for the classification of detected clones as per their re-factoring opportunities. Svajlenko et al. [49] used unsupervised machine learning based approach to cluster similar code clones. Kapser and Godfrey [48] classified the detected clones based on their location with respect to one another in the hierarchy files and directories. However, it is noticeable from the above works that the total number of clones to be manually analyzed for the validation still remain the same. The overall result of such code clone classification or comprehension techniques can be improved significantly by adding a machine learning based automatic validation process.

Several clone visualization techniques have also been proposed to organize large amount of detected clones [50] such as, scatter plot of the code clones [8], Hasse diagram [51], an aspect browser-like view [19], hierarchical graphs of detected clones [12] and so on. Note that, our proposed work can aid in further management and understanding of the clones by adding an extra dimension (e.g., predicted true or false positive) to these existing visualization techniques. Besides, our presented insights on human validation pattern can help better understand clone detection problem and facilitate on further improvement of machine learning models such as ANNs, Deep Learning and so on for clone detection tools [52]. Wang et al. [53] proposed a search based method find the optimal configurations of code clone detection tools to tune their corresponding reported clones. While their work is on establishing a general agreement on code clone detection tools' optimal configurations [18], our work focuses on a different goal for automating the clone validation process.

Besides, researchers often find it challenging to evaluate any tools or techniques for clone detection due to the lack of enough validated code clone benchmark. Because building such benchmarks often contain possible threats to validity due to unavoidable human errors and need a huge amount of manual validation work. For example, Bellon et al. [10] created one such benchmark by validating 2% of the union of six clone detectors for eight subject systems that required 77 hours of manual efforts. Svajlenko et al. created a benchmark of 78 thousands snippets that also reports huge amount of manual effort requirement with 514 hours of manual validation efforts of nine human judges [7]. So, the trained machine learning model can be used to aid in creation of user specific validated clone sets.

VIII. Conclusion

In this paper, we proposed a machine learning based approach for automatic code clone validation. The method learns to predict tasks or user-specific code clone validation patterns. A prototype system [23], which was developed as a proof of concept of the proposed method, showed promising results in comparison to related existing methods for code clone validation. We also evaluated the proposed system with different users, clone detection tools, artificially created code clones, open source projects and so on, and presented our several insights such as performance of different machine learning methods, data distribution analysis and so on for the code clone validation problem. In future, we will explore the enhancement of existing clone visualization techniques with the aid of automatic code clone validation responses. Our future work plan also includes scaling up the proposed method for large scale code clone managements as we have presented in the vision of the proposed method.

References

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

[2] Chanchal K. Roy and James R. Cordy. An empirical study of function clones in open source software. In *Reverse Engineering, 2008. WCRE'08. 15th Working Conference on*, pages 81–90. IEEE, 2008.

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

[4] Cory J Kapser and Michael W Godfrey. Supporting the analysis of clones in software systems. *Journal of Software: Evolution and Process*, 18(2):61–82, 2006.

[5] Cory Kapser and Michael W Godfrey. " cloning considered harmful" considered harmful. In *Reverse Engineering, 2006. WCRE'06. 13th Working Conference on*, pages 19–28. IEEE, 2006.

[6] Elmar Juergens, Florian Deissenboeck, Benjamin Hummel, and Stefan Wagner. Do code clones matter? In *Software Engineering, 2009. ICSE 2009. IEEE 31st International Conference on*, pages 485–495. IEEE, 2009.

[7] Jeff Thomas Svajlenko et al. *Large-Scale Clone Detection and Benchmarking*. PhD thesis, University of Saskatchewan, 2018.

[8] Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. CCFinder: a multilinguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering*, 28(7):654–670, 2002.

[9] Ekwa Duala-Ekoko and Martin P Robillard. Tracking code clones in evolving software. In *Software Engineering, 2007. ICSE 2007. 29th International Conference on*, pages 158–167. IEEE, 2007.

[10] Stefan Bellon, Rainer Koschke, Giulio Antoniol, Jens Krinke, and Ettore Merlo. Comparison and evaluation of clone detection tools. *IEEE Transactions on software engineering*, 33(9), 2007.

[11] Chanchal K Roy and James R 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*, pages 172–181. IEEE, 2008.

[12] Lingxiao Jiang, Ghassan Misherghi, Zhendong Su, and Stephane Glondu. Deckard: Scalable and accurate tree-based detection of code clones. In *Proceedings of the 29th international conference on Software Engineering*, pages 96–105. IEEE Computer Society, 2007.

[13] Robert Tairas and Jeff Gray. Phoenix-based clone detection using suffix trees. In *Proceedings of the 44th annual Southeast regional conference*, pages 679–684. ACM, 2006.

[14] Rainer Koschke, Raimar Falke, and Pierre Frenzel. Clone detection using abstract syntax suffix trees. In *Reverse Engineering, 2006. WCRE'06. 13th Working Conference on*, pages 253–262. IEEE, 2006.

[15] Zhen Ming Jiang and Ahmed E Hassan. A framework for studying clones in large software systems. In *Source Code Analysis and Manipulation, 2007. SCAM 2007. Seventh IEEE International Working Conference on*, pages 203–212. IEEE, 2007.

[16] Iman Keivanloo, Feng Zhang, and Ying Zou. Threshold-free code clone detection for a large-scale heterogeneous java repository. In *Software Analysis, Evolution and Reengineering (SANER), 2015 IEEE 22nd International Conference on*, pages 201–210. IEEE, 2015.

[17] Alan Charpentier, Jean-Rémy Falleri, David Lo, and Laurent Réveillère. An empirical assessment of bellon's clone benchmark. In *Proceedings of the 19th International Conference on Evaluation and Assessment in Software Engineering*, page 20. ACM, 2015.

[18] Jiachen Yang, Keisuke Hotta, Yoshiki Higo, Hiroshi Igaki, and Shinji Kusumoto. Classification model for code clones based on machine learning. *Empirical Software Engineering*, 20(4):1095–1125, 2015.

[19] Robert Tairas and Jeff Gray. An information retrieval process to aid in the analysis of code clones. *Empirical Software Engineering*, 14(1):33–56, 2009.

[20] Zhenmin Li, Shan Lu, Suvda Myagmar, and Yuanyuan Zhou. Cp-miner: Finding copy-paste and related bugs in large-scale software code. *IEEE Transactions on software Engineering*, 32(3):176–192, 2006.

[21] David Lo, Lingxiao Jiang, Aditya Budi, et al. Active refinement of clone anomaly reports. In *Proceedings of the 34th International Conference on Software Engineering*, pages 397–407. IEEE Press, 2012.

[22] Yoshiki Higo, Shinji Kusumoto, and Katsuro Inoue. A metric-based approach to identifying refactoring opportunities for merging code clones in a java software system. *Journal of Software: Evolution and Process*, 20(6):435–461, 2008.

[23] G. Mostaeen, Jeffrey Svajlenko, Banani Roy, Chanchal K. Roy, and K. Schneider. A Prototype System for Clone Validation. http://p2irc-cloud.usask.ca/ccv (pls. contact on issues).

[24] Ambient Software Evoluton Group. IJaDataset 2.0. http://secold.org/projects/seclone.

[25] James R Cordy and Chanchal K Roy. The NICAD clone detector. In *Program Comprehension (ICPC), 2011 IEEE 19th International Conference on*, pages 219–220. IEEE, 2011.

[26] Nils Göde and Rainer Koschke. Incremental clone detection. In *Software Maintenance and Reengineering, 2009. CSMR'09. 13th European Conference on*, pages 219–228. IEEE, 2009.

[27] Hitesh Sajnani, Vaibhav Saini, Jeffrey Svajlenko, Chanchal K Roy, and Cristina V Lopes. Sourcerercc: Scaling code clone detection to big-code. In *Software Engineering (ICSE), 2016 IEEE/ACM 38th International Conference on*, pages 1157–1168. IEEE, 2016.

[28] Jeffrey Svajlenko and Chanchal K Roy. Cloneworks: a fast and flexible large-scale near-miss clone detection tool. In *Proceedings of the 39th International Conference on Software Engineering Companion*, pages 177–179. IEEE Press, 2017.

[29] Jeffrey Svajlenko and Chanchal K Roy. Fast and flexible large-scale clone detection with cloneworks. In *Software Engineering Companion (ICSE-C), 2017 IEEE/ACM 39th International Conference on*, pages 27–30. IEEE, 2017.

[30] Weka. Open Source Machine Learning Tools Collection. https://www.cs.waikato.ac.nz/ml/weka/.

[31] James R Cordy, Charles D Halpern-Hamu, and Eric Promislow. Txl: A rapid prototyping system for programming language dialects. *Computer Languages*, 16(1):97–107, 1991.

[32] Douglas Crockford. The application/json media type for javascript object notation (json). 2006.

[33] Christian Robert. Machine learning, a probabilistic perspective, 2014.

[34] George H John and Pat Langley. Estimating continuous distributions in bayesian classifiers. In *Proceedings of the Eleventh conference on Uncertainty in artificial intelligence*, pages 338–345. Morgan Kaufmann Publishers Inc., 1995.

[35] Nir Friedman, Dan Geiger, and Moises Goldszmidt. Bayesian network classifiers. *Machine learning*, 29(2-3):131–163, 1997.

[36] Wai Lam and Fahiem Bacchus. Learning bayesian belief networks: An approach based on the mdl principle. *Computational intelligence*, 10(3):269–293, 1994.

[37] Leo Breiman. Random forests. *Machine Learning*, 45(1):5–32, 2001.

[38] Ross Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann Publishers, San Mateo, CA, 1993.

[39] Jeffrey Svajlenko, Chanchal K Roy, and James R Cordy. A mutation analysis based benchmarking framework for clone detectors. In *Software Clones (IWSC), 2013 7th International Workshop on*, pages 8–9. IEEE, 2013.

[40] Flask. A Microframework for Python based on Werkzeug, Jinja 2. http://flask.pocoo.org/.

[41] CouchDB. Apache CouchDB - NoSQL Database System. http://couchdb.apache.org/.

[42] Jeffrey Svajlenko, Judith F Islam, Iman Keivanloo, Chanchal K Roy, and Mohammad Mamun Mia. Towards a big data curated benchmark of inter-project code clones. In *Software Maintenance and Evolution (ICSME), 2014 IEEE International Conference on*, pages 476–480. IEEE, 2014.

[43] Md Sharif Uddin, Chanchal K Roy, and Kevin A Schneider. Simcad: An extensible and faster clone detection tool for large scale software systems. In *Program Comprehension (ICPC), 2013 IEEE 21st International Conference on*, pages 236–238. IEEE, 2013.

[44] Simian. Code Clone Detection Tool. http://www.redhillconsulting.com.au/products/simian/.

[45] Warren Toomey. Ctcompare: Code clone detection using hashed token sequences. In *Software Clones (IWSC), 2012 6th International Workshop on*, pages 92–93. IEEE, 2012.

[46] Andrian Marcus and Jonathan I Maletic. Identification of high-level concept clones in source code. In *Automated Software Engineering, 2001.(ASE 2001). Proceedings. 16th Annual International Conference on*, pages 107–114. IEEE, 2001.

[47] Adrian Kuhn, Stéphane Ducasse, and Tudor Gírba. Semantic clustering: Identifying topics in source code. *Information and Software Technology*, 49(3):230–243, 2007.

[48] Cory Kapser and Michael W Godfrey. Aiding comprehension of cloning through categorization. In *Software Evolution, 2004. Proceedings. 7th International Workshop on Principles of*, pages 85–94. IEEE, 2004.

[49] Jeffrey Svajlenko and Chanchal K Roy. A machine learning based approach for evaluating clone detection tools for a generalized and accurate precision. *International Journal of Software Engineering and Knowledge Engineering*, 26(09n10):1399–1429, 2016.

[50] Minhaz F Zibran and Chanchal K Roy. The road to software clone management: A survey. *Dept. Comput. Sci., Univ. of Saskatchewan, Saskatoon, SK, Tech. Rep*, 3, 2012.

[51] J Howard Johnson. Visualizing textual redundancy in legacy source. In *Proceedings of the 1994 conference of the Centre for Advanced Studies on Collaborative research*, page 32. IBM Press, 1994.

[52] Liuqing Li, He Feng, Wenjie Zhuang, Na Meng, and Barbara Ryder. Cclearner: A deep learning-based clone detection approach. In *Software Maintenance and Evolution (ICSME), 2017 IEEE International Conference on*, pages 249–260. IEEE, 2017.

[53] Tiantian Wang, Mark Harman, Yue Jia, and Jens Krinke. Searching for better configurations: a rigorous approach to clone evaluation. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, pages 455–465. ACM, 2013.