On the Use of Machine Learning Techniques in Automatic Validation of Code Clones

Anonymous Author(s)

***Abstract*—A code clone is a pair of code fragments, within or between software systems that are similar. Since code clones can reduce the maintainability of software system, several code clone detection techniques and tools have been proposed and studied over the last decade. However, the clone detection tools are not always perfect and their clone detection reports often contain a number of false positives. This often means the clones must be manually inspected before analysis, to remove those false positives from consideration. 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 approach for automating the validation process. The possible code clone pairs output from several existing code clone detection tools are taken, manually validated and several features are extracted from them to train an Artificial Neural Network by the proposed approach. The trained algorithm can then be used to automatically validate clones without human inspection. Thus the proposed approach can be used to remove the false positive clones from the detection results to improve the precision of existing clone detectors, evaluate existing clone benchmark datasets, or even be used to build new clone benchmarks and datasets with a minimum of effort. In an experiment with clones detected by several clone detectors in several different software systems, we found our approach has an accuracy of 87.4% when compared against the manual validation by multiple expert judges.**

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], [2] : 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 identified 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. 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 [3], [4], [5]. Though code cloning is often done intentionally to accelerate the development process and also not all code clones are harmful [6], 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. For example, it becomes very difficult to carry out a consistent changes to all the cloned code fragments throughout the software system. This inconsistent changes to the corresponding duplicated code fragments are often responsible for creation of new software bugs. In addition to creation of new bugs code cloning also becomes one of the main reasons for bug propagation when programmers copy-and-paste a buggy code fragment throughout the software system for implementing similar functionalities. Detection of such code clones can therefore accelerate the maintenance task of any software systems remarkably [7]. Besides, exploiting the similarities of the detected code clones also helps to better understand and improve the overall software design [8].

At least 70 Clone Detection Tools (CDTs) and techniques have been proposed and developed to automate the clone detection process, as a result of extensive research in this specific area over the last decade [9], [10], [11], [12], [13], [14], [15]. These tools return a list of possible code clone pairs available in a given software system. Except Type 1, the other types of code clones (Type 2, 3 and 4) undergoes different changes over time and gets too complicated to be detected by a simple string matching algorithm. For example the identifiers or functions names may be changed, some code fragments 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 etc. All these modifications over times make the searching problem much more complicated and to handle those complex source code structures while still detecting all possible code clone pairs, the tools undergoes a lot of generalization of the original source codes like pretty-printing [13], normalization of the identifiers [9], [13], forming syntax tree [16] of the code fragments and so on just to name a few.

As a result of this complex searching problem and necessary generalization/normalization of the source code, the clone detection algorithms often report false positive clones. These are pairs of code fragments that are not similar, are only coincidentally similar, or are otherwise considered not a valid clone by the user [17]. 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 or software systems [18], [19], [20]. For these reasons programmers often need to manually validate if the result is a true clone or not before using this information for the given specific scenarios like: source code refactoring or other software maintenance task. This manual validation process even becomes a hindering factor as software system evolves in size over time. Because in that case programmers often find it challenging to extract the actual true positive clones they are looking for from those large set of reported possible code clone pairs by clone detection tools. For example some previous research shows that JDK 1.4.2 contains 204 K LoC reported code clone which is 8% of the total lines of code [17], [21]. 15% of the total lines of code of the Linux kernel has been reported as code clone which is 122 K LoC [22]. Both of the above scenarios on number of reported possible code clones by clone detection tools illustrate the huge amount of manual validation work necessary before using the code clone information. Besides the clone detection algorithms of the tools usually work in general irrespective of the specific system requirements or user preferences. So in the best case even if a detection tool returns all the true positive clones only, many of the clone pairs might be unimportant for the given specific system requirements or the programmers preferences as the tool fails to predict and return results accordingly [17]. Mining those code clones of interests from the tool generated report is often a time consuming task and thus reduce the usability of code clone detection tools. The scenario even gets worse with the increase of software project in size.

In this paper we studied performance and result qualities of different machine learning algorithms in an attempt to automate the validation process of such subjective clones. We propose an automatic validation process using machine learning algorithm from our study and findings. We also propose a cloud based architecture to ensure compatibility of the proposed method with any of the existing CDTs. We got promising result in the comparative study with related existing works for code clone validation.

II. Background

In this section, we first introduce some of the existing systems for user specific code clone validation along with related works and then we present available scopes of improvements and motivations for our work.

A. *Traditional Approaches for Code Clone Classification*

Though 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. [20] studied on the similar problem for user specific code clone classification in their work entitled as - 'Filter for Individual user on code Clone Analysis'- FICA. The code clone classification in FICA is done by token sequence similarity analysis using 'Term-Frequency - Inverse Document Vector'- (TF-IDF) vector. For training some of the reported code clone pairs from CDTs are marked as True or False positive clones by the users. So, the whole training set M, is divided into two sets of clone sets - M_t and M_f such that $M_t \cap M_f = \emptyset$ and $M_t \cap M_f = M$. Tokens are then extracted from both the clone sets to produce n-gram (considered N=3 for the study) of token sequences. Defining term t as an n-gram of token sequence and document d as a clone set, the Term Frequency (TF) are calculated as Eq. 1.

$$TF(t, d) = \frac{t : t \in d}{|d|} \qquad (1)$$

Similarly defining documents D, as all the clone sets of a project or software systems of considerations, the Inverse Document Frequency (IDF) and TF-IDF vector are calculated using Eq. 2 and Eq. 3 respectively.

$$IDF_D(t) = \log \frac{|D|}{1 + |d \in D : t \in d|} \qquad (2)$$

$$\overrightarrow{TF\text{-}IDF_D(d)} = [TF\text{-}IDF_D(t, d) : \forall t \in d] \qquad (3)$$

Using TF-IDF for the two clone sets a and b, cosine similarity $CosSim_D(a, b)$ are then calculated for a set of documents D. The probability score for an unmarked clone set c, for being in clone set M_t or M_f are then calculated using the Eq. 4.

$$P_{M_x}(c) = \frac{\sum_{\forall m \in M_x} CosSim_{M_x}(c, m).w(m)}{\sum_{\forall m \in M_x} w(m)} \qquad (4)$$

Here, $w(m)$ are assigned weight in the range $[0, 1]$, in case of numerical range marking of the clones. However, in case boolean marking (i.e. either M_t or M_f) of clones, $w(m)$ is uniformly set to 1. Besides, for the improvement of the model, FICA optionally takes user feedback iteratively over time to populate training set M_t and M_f. 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 the study.

B. *Related Works*

In recent years a number of research works has also been done for CDTs' reported clone classification or comprehension. For example, Tairas et. al. [21] broadly classified the existing code clone comprehension techniques into two categories. The first category of the techniques does the classification of the detected code clones based on certain properties: location of the clones with respect to one another in the hierarchy files and directories [23], type similarities (all possible Type 1 clones grouped together and so on for Type 2, 3 and 4) of the detected clones [12], Latent Semantic Indexing (LSI) on the identifiers of clones [21] and so on. Besides, some machine learning algorithms have also been applied to group the detected clones: token sequence similarities of the clones have been analyzed to categorize them [20], applying unsupervised machine learning algorithm to create clone clusters [24] and so on. On the other hand the second category of clone comprehension techniques works based on their visual representation: scatter plot of the code clones [9], an aspect browser-like view [21], hierarchical graphs of detected clones [14] and so on. These classification and visualization techniques can make the organization and maintenance of code clone much easier.

C. *Motivation*

We can notice that the total number of clones to be manually analyzed for validation still remain the same. The overall result of such code clone comprehension techniques can be improved significantly by adding an automatic validation process that uses machine learning approach to learn to validate according to the specific system and user over time.

Besides, researchers often find it challenging to evaluate any tools or techniques on 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 huge amount of manual validation work. For example Bellon et al. [12] 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. [25] created a benchmark of true positive clones that also reports hours of manual validation efforts. So, the trained machine learning model can be used to aid in creation of user specific validated clone sets.

The proposed method works as a layer on top of the reported possible code clone pairs generated by existing clone detection tools. Initially some user validated clones are fed to the system for learning the validation behaviour of the specific system or programmer. The proposed method extract several features right from the reported code clone pairs of source code fragments. Once the training phase with validated code clone pair is completed the system is given unknown or test code clone pairs to validate. The proposed system extract the exact same features from the test code clone pairs and feeds them to the trained machine learning algorithm where it gives the validation response. The programmers feedback result can optionally be given to the system to update and improve its prediction rule. This gives the proposed method an opportunity to improve and learn the programmers preferences even better with time and experiences. Besides the proposed method can optionally be tuned to control the validated result based on the importance of the code clone pairs. That is a programmer can choose to get all the code clone pairs reported by a clone detection tools or only those code clone pairs having some particular importance as per their preferences. This gives the programmer a flexibility on selecting the number of validated clones for some given scenarios.

This study is aimed to answer the following 3 research questions:

- **RQ 1** : Can the manual code clone validation process be assisted via machine learning?
- **RQ 2** : Can the machine learning technique predict a programmers validation behaviour to improve itself over time and experiences?
- **RQ 3** : Does the proposed machine learning based validation method works across different clone types and clone detection tools?

III. Dataset Description

As the machine learning algorithm tries to recognize any available underlying pattern in the given dataset, it is very important how we choose the dataset and which features we extract out of it for training and testing of the system. For example selecting a smaller or undiversified dataset can make the algorithm biased and fails to generalize all the other different types of clones. So to get generalization in validating different types of code clones by the system we have chosen to use relatively bigger and diverse dataset. Besides we also considered clones reported by different existing code clone detection tools from those multiple open source projects.

We have divided the discussion of this section on dataset description in three parts. In Section III-A we discuss about the sources of data that we are using, next Section III-B contains the discussion on high level details of the dataset that was used to train and test the system and finally in Section III-C we discuss about different statistical summary and underlying distribution we found studying the dataset.

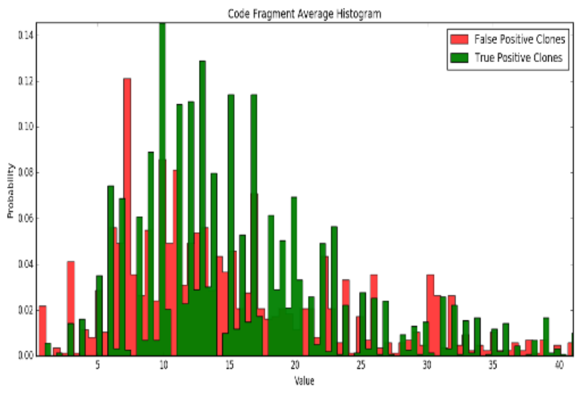


Fig. 1: Histogram of Code Fragment Average

A. Data Source

For training, we used clones from IJaDataset 2.0 [26], 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 [27], Deckard [14], iClones [28], CCFinderX [9] and SourcererCC [29] were used to detect clones separately out of the benchmark. Randomly 400 clone pairs were then selected and manually validated from each of the five clone detection tools separately. We have chosen to work on these dataset because a good number of recent research works on code clones has been carried out on these open source projects and thus we can have a common ground for evaluating our proposed approach.

B. High Level Details of the Data Set

Reports obtained from any of the existing clone detection tools on possible code clone pairs are given as input to the proposed method for validation purpose. Several code clone detection tools were run on the used data source to find the corresponding reports for the possible code clone pairs. The code clone pairs were then manually validated for the training phase of the proposed method. As some recent research shows that the clone validation decision in some scenario depends on users perspective [18], that is given a possible code clone pair to validate some judges might decide it to be a true positive clone pairs where others might say the opposite (especially in case of Type 3 and Type 4 clones). So to give this generalization to the proposed method the whole set of code pairs were split into 5 parts to be validated by 5 different programmers independently. This manual validation decision along with the corresponding possible code clone pairs are given as input to the proposed method for the training phase.

C. Studying Underlying Data Distribution

Out of those manually validated clones we extracted different features that are fed to machine learning algorithms to learn to validate automatically based on those features on new test clone pairs. In this section we will discuss on different distribution and statistical studies and behaviors of some of the extracted features.

For every code clone pairs detected by clone detection tools we found the similar code fragments for a clone pairs.

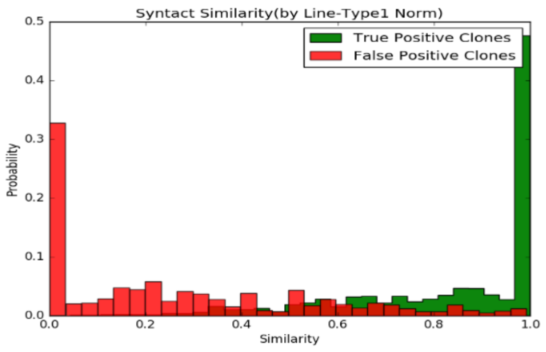


Fig. 2: Histogram of Syntactical Similarity by Line (Type 1 Norm.)

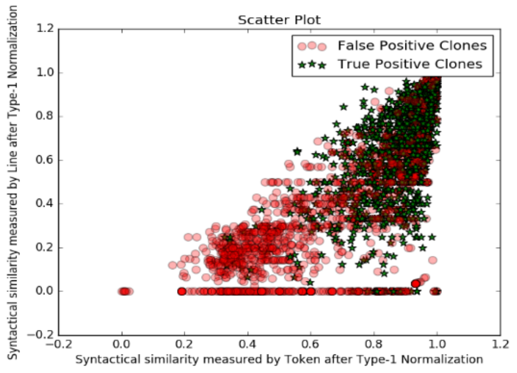


Fig. 4: Syntactical Sim. by Line vs Token (Type 1 Norm.)

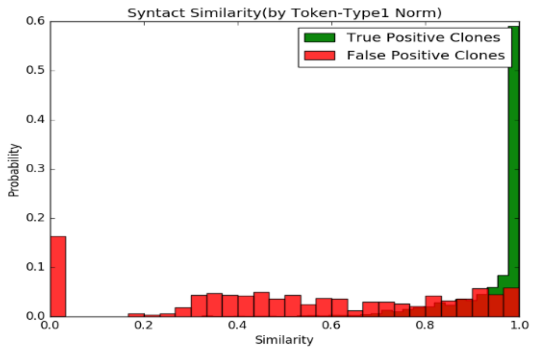


Fig. 3: Histogram of Syntactical Similarity by Token (Type 1 Norm.)

These are the similar code fragments for which the tools decided them to be possible code clone pair. For the largest and smallest similar code fragments the corresponding sizes were calculated as α and β respectively. Our intuition was if the code fragments are significantly different in sizes, validator may be more likely to mark them as false positive. So we took the difference, $(\alpha - \beta)$ as one possible feature. However for a clone that is small versus a clone that is large might have different consideration. 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. We analyzed this 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 1, shows the distribution of the average code clone fragment feature for the true positive and false positive clone classes. From the figure we can notice that, the average code fragment size shows much randomness, both for true positive and false positive clones. The distribution of this feature almost overlaps on one another for the two classes: true positive and false positive code clones. This overlapping pattern suggests that this feature provides very minimal information about the two classes and thus yields a very low contribution score for training the machine learning algorithm for validation.

Besides for extracting some other features we have normalized the code clone pairs by 3 levels of Normalization namely: Type 1, Type 2 and Type 3. Then for each level of normalizations syntactical similarity was measured by lines and by tokens for the clone pairs resulting 6 different possible features. To view any underlying distribution of the features their normalized histogram were plotted both for true positive and false positive clones. Figure 2, shows one of such plottings that is based on the similarity measured by lines after Type 1 code normalization. From the figure it is noticeable that the distribution of the feature is comparatively better than the average code fragment line feature in terms of validation. Though the distribution for true positive and false positive clones are not completely linearly separable with this feature but still the two classes are somewhat distinguishable. The distribution indicates a possible better contribution score for validation prediction than the average clone fragment sizes. Figure 3, also shows somewhat similar results in case of Syntactic Similarity measured by tokens after Type 1 Normalization of the source codes.

We also carried out several studies to find out any underlying relationship between different features for possible clustering of the two clone classes. For example we tried to figure out if there is any underlying relationship 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 4, is one of such figures that shows the scatter plot on syntactical similarity measured by line versus tokens after Type 1 Normalization of the code clone pairs for true positive and false positive clone classes. However these analysis did not show any distinguishable clusters for the two classes.

As machine learning algorithms try to recognize any underlying pattern available on the working dataset, the detail analysis on the dataset and possible features are necessary for selecting the right machine learning algorithm and features. So from this distribution analysis on different possible features for code clones provides us information about their importance and contribution for clone validation. This analysis provides a clearer view of the data distribution and thus helps picking the appropriate machine learning algorithm and corresponding features for the algorithm. From several analysis on the data distribution we tried to find out the features that has comparatively more distinguishable distribution and provides more contribution for the two classes true positive and false positive clones. Table I, shows a feature set ranked on possible contribution score based on our analysis study. The corresponding distribution mean differences for the two classes also somewhat indicates the separability for the classification.

IV. Proposed Approach

The data distribution study (Section III-C) exhibits that the two classes are not linearly separable or does not contain any distinguishable clone clusters. The study shows that the clone classes are not straightforward or linear for separation, for example by usage of single or multiple thresholds [18] in terms of the features by CDTs [9], [10]. On the other hand, somewhat recognizable distribution for the two classes for some of the features as discussed in Section III-C, indicates the possibility of having complex underlying patterns or rules available in the data set that can contribute for the classification problem of clone validation. So our intuition was that, if we can improve the tool generated report on code clone by exploiting those complex patterns and the user preferences by mapping the extracted feature set to the corresponding subjective clone validation problem. So, in an attempt for such improvement on clone report we applied and studied different machine learning algorithms on the open source software projects. Figure 5, shows a high level workflow (which has been extended for the cloud based architecture as discussed in Section IV-B in details) of the training phase of the proposed method.

To give more generality to the machine learning model we used five different clone detection tools (NiCad [27], Deckard [14], iClones [28], CCFinderX [9] and SourcererCC [29]) for detecting possible software clones from the open source projects as shown in Step 2 (Figure 5). The code clone pairs were then manually validated for the training phase of the proposed method. As some recent research shows that the clone validation decision in some scenario depends on users perspective [18], that is given a possible code clone pair to validate some judges might decide it to be a true positive clone pairs where others might say the opposite (especially in case of Type 3 and Type 4 clones). So to minimize the biasness in the proposed method the whole set of code pairs were split into 5 parts to be validated by 5 different programmers independently (Step 4). In the next step, we then extracted previously analyzed features from those manually validated clone pairs for training. TXL was used for normalization and several other pre-processing step on the source code

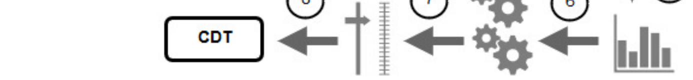


Fig. 5: Workflow of the proposed method

before finding the features. We used Java then on those pre-processed and normalized source code to extract features. We selected several features that shows better result based on the study of data distribution. We selected 8 such features (Table I): Syntactical similarity both of line and token after Type-1, Type-2 and Type-3 normalization separately, Number of unmatched braces for the code fragments and Number of intersected functions between these possible code clone pairs. The extracted features of the manually validated code clones are fed into the supervised machine learning algorithms for training the model in Step 6. To study the comparative classification performances we applied 13 different machine learning algorithm. The algorithms were tested using 10-fold cross validation method. On an average we found promising classification accuracy for some of those algorithms. The detail comparison study of the algorithms has been discussed in IV-A. Finally the trained model are then used for predicting the subjective code clone validation by different CDTs. In addition to using the trained model locally for validation classification, we also propose a cloud based architecture for different additional advantages. We have discussed about the architecture and the prototype implementation in detail in Section IV-B.

A. Comparative Study on Machine Learning Algorithms for Clone Classification

As we have presented the workflow of the proposed method in the above discussion, we need to use supervised machine learning algorithm that learns to classify the user specific clone validation (i.e. Step 6). The supervised classification algorithm will be trained on the manually validated datatset $D = \{(x_1, y_1), (x_2, y_2)...(x_m, y_m)\}$, for $x_i \in \mathbb{R}^n$ and $y_i \in \mathbb{R}^l$, where n and l represent the extracted clone feature set and clone validation labels respectively. The machine learning algorithm is then trained on dataset D, to learn a function f, such that f can map from $\mathbb{R}^n$ to $\mathbb{R}^l$.

We investigated the classification performance using different machine learning algorithms as to the best of our knowledge, we could not find any other previous research works that directly focused on user specific clone validation using such extracted clone features to target validation of all 3 different types of clones. The most relevant works we found used some trivial sequence matching (for example: TF-IDF, token sequence matching and so on) algorithms instead and failed to validate beyond Type 2 clones (i.e. FICA [20]). In the comparative study of used 13 different algorithms, we got accuracies within a range of 76% to 87%. Due to the page limitation, it is not possible to show all the experimental results here in details. So, we presented the obtained algorithm accuracies in Figure 6 and summary of the result qualities in Table II. We got the best result from Backpropagation Neural Network, which we discuss in details in the following.

From the training dataset D, for $x_i \in \mathbb{R}^n$ in the input layer, $y_i \in \mathbb{R}^l$ in the output layer and one hidden layer with k nodes, the ANN learns the following function:

$$f(x) = \sigma(W_{ho}^{\intercal} \cdot \sigma(W_{ih}^{\intercal} \cdot x + \theta_h) + \theta_0) \quad (5)$$

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. We used softmax

TABLE I: Selected Features Based on Distribution Analysis

Feature	Distribution Mean Difference	Feature Description
Line Sim. (Type-1 Norm.)	0.3998	Syntactical similarity measured by line after Type-1 Normalization
Line Sim. (Type-2 Norm.)	0.3701	Syntactical similarity measured by line after Type-2 Normalization
Line Sim. (Type-3 Norm.)	0.3602	Syntactical similarity measured by line after Type-3 Normalization
Token Sim. (Type-2 Norm.)	0.3447	Syntactical similarity measured by Token after Type-2 Normalization
Token Sim. (Type-1 Norm.)	0.3105	Syntactical similarity measured by Token after Type-1 Normalization
Token Sim. (Type-3 Norm.)	0.2537	Syntactical similarity measured by Token after Type-3 Normalization
Function Intersected	0.2364	Total Number of functions intersected by the code fragments
Unmatched Braces	0.2078	Total number of unmatched braces across both code fragment

activation function for the output layer. The learned function $f(x)$, is then used to predict the clone validation for the new test feature set x_t from the corresponding probability values:

$$\hat{y}_t = f(x_t) = (P[y_t = (1,0)], P[y_t = (0,1)]) \qquad (6)$$

where, $P[y_t = (1,0)]$ denotes the probability of the test code clone with feature x_t to be true positive, λ. So, for a preset user preference value $\gamma[0,1]$, the proposed approach decides the test code clone as True Positive if $\lambda \geq \gamma$ and False Positive otherwise.

For the training phase the Neural Network was run with $k = 107$, for a number of epochs until it converges with a maximum limit of 1000 epochs. The model was trained and tested using 10-fold cross validation. Figure 7, shows the accuracy of the method as it converges versus the epochs (averaged for each of the 10-fold validation). The Neural Network converged within a range of 500 to 600 epochs, giving an accuracy of 87.4%.

In the proposed method, users can set the decision threshold $\gamma[0,1]$ to tune the validation output quality. On setting this γ value towards its upper limit will return only those clone pairs having higher probability of being true positive clones. Thus most of the returned results are expected to be true code clone pairs. Oppositely one can decrease the value of γ to get more result from the clone detection tool. That is this decision threshold can be helpful for the users to tune the result quality as per the requirements.

To analyze the output quality across different values of we plotted the ROC curve, which is shown in Figure 8. The calculated Area Under the Curve (AUC) for the ROC curve is

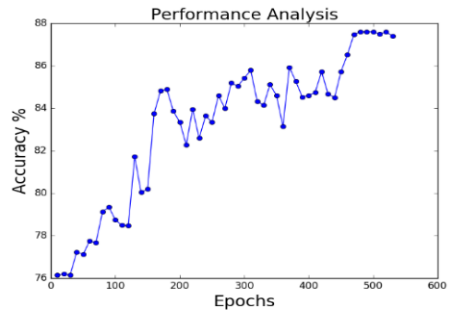


Fig. 7: Avg. Accuracy for 10-fold cross validation (as the algorithm converges vs epochs).

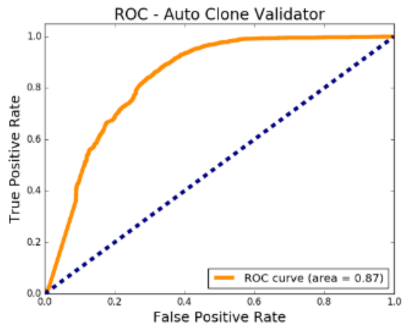


Fig. 8: ROC - Curve for validation by the method

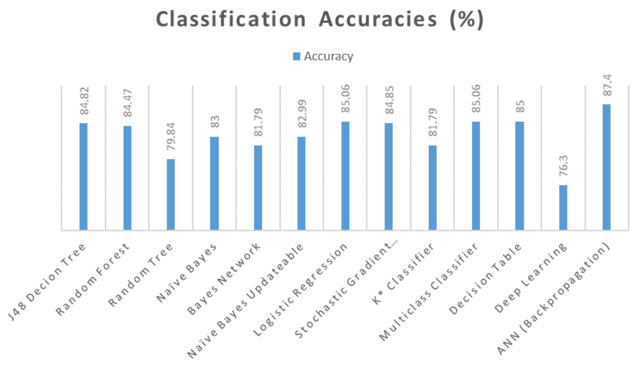


Fig. 6: Accuracy Comparison of the Methods across Different Software Systems.

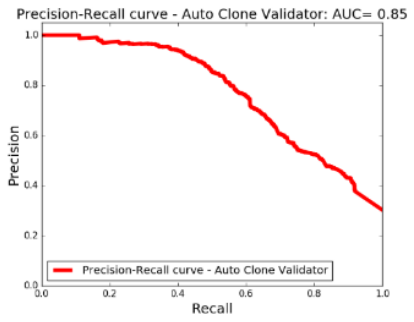
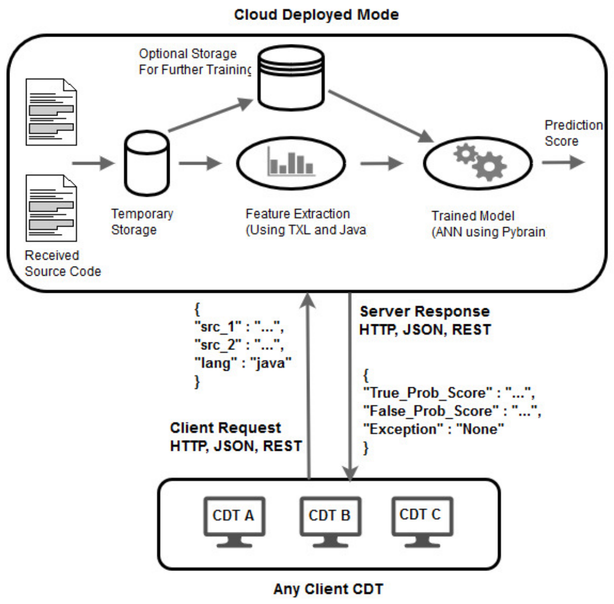


Fig. 9: PR - Curve for validation by the method

TABLE II: Classification Result Quality for Different Machine Learning Algorithms

Classifiers	TP Rate	FP Rate	Precision	Recall	F-Measure	ROC-area
J48 Decision Tree	0.848	0.291	0.849	0.848	0.840	0.803
Random Forest	0.845	0.254	0.841	0.845	0.841	0.892
Random Tree	0.789	0.275	0.799	0.798	0.799	0.793
Nave Bayes Classifier	0.830	0.332	0.831	0.830	0.818	0.828
Bayes Network	0.818	0.266	0.815	0.818	0.816	0.830
Nave Bayes Updateable	0.830	0.332	0.831	0.830	0.818	0.828
Logistic Regression	0.851	0.292	0.852	0.851	0.842	0.845
Stochastic Gradient Descent	0.849	0.308	0.854	0.849	0.838	0.770
K* Classifier	0.818	0.287	0.813	0.818	0.814	0.848
Multiclass Classifier	0.851	0.292	0.852	0.851	0.842	0.845
Decision Table	0.850	0.292	0.852	0.850	0.841	0.845


Cloud Deployed Mode
Optional Storage For Further Training
Received Source Code
Temporary Storage
Feature Extraction (Using TXL and Java
Trained Model (ANN using Pybrain
Prediction Score
Client Request HTTP, JSON, REST
Server Response HTTP, JSON, REST
CDT A
CDT B
CDT C
Any Client CDT

```
{
"src_1" : "...",
"src_2" : "...",
"lang" : "java"
}
```

```
{
"True_Prob_Score" : "...",
"False_Prob_Score" : "...",
"Exception" : "None"
}
```

Fig. 10: Cloud Model for Compatibility with Existing Code Clone Detection Tools

found to be 0.87. From the ROC curve the proposed method can recommend the value to the users by default that gives the best result in terms of the ratio of the true positive and false positive ratio in the output result while training. Besides Figure 9, shows the Precision-Recall curve for the proposed method for varying γ values. In case of Precision-Recall curve the AUC found to be 0.85.

B. Cloud Architecture for Clone Classification

Machine learning model generalization and performance depends on the quality and quantity of the training data set. With relatively more data set the machine learning models are expected to perform better. As the proposed method can work on top of any clone detection tool generated results, to increase the usability we propose a cloud based architecture. Figure 10, gives an overview of the architecture. Reported code clones from a target CDT are send to the server for validation using HTTP request. On receiving the subject clones to validate, the required features are extracted to create feature set x_t, which is then used by the learned function to get the probability score $f(x_t)$. The corresponding scores are then send back to the CDT for displaying validated result in the user end.[1] Some of the advantages of the cloud deployment of the model can be discussed as the following:

1) Compatibility with Existing Clone Detection Tools: All the platforms and implementation dependencies can be abstracted from the cloud implementation. For example existing different clone detection tools are developed for different platforms specifically or with different programming languages. Providing a common way that is understandable by all the tools irrespective of their implementation varieties can thus improve the usability of the proposed method to a great extent.

2) Improvement in the Training Phase: Getting enough training data set or time by individual user for the machine learning algorithm can often be challenging. On the other hand, for the cloud based model deployment the user can take advantage of the trained model. Triggering a new model learning on train data set is also simpler and involves a single request to the cloud. Additionally user can also choose among different trained models (i.e. ANN, Decision Tree and so on) for better convergence with their decision.

3) Opportunities for Higher Processing Power: Cloud based model deployment opens up the possibilities for higher processing power with cluster or distributed computing in the future. Thus the higher processing advantages for big data set of clone validation is possible with even from relatively low processing power user end devices.

V. Result Analysis

Automatic clone validation can contribute in the clone analysis across different scenarios and requirements starting from starting from smaller to large scale software system. For this reason we were interested on evaluating the system across different environmental set up: with several clone detection tools, users, software systems and so on.

A. 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

[1]We implemented a prototype of the proposed system. All the source codes and required resources are available at for replication and research purposes https://github.com/pseudopixels/codeCloneVal

clones as true or false positive varies significantly from person to person [18], [19]. Because evaluation result against such true positive or false positive already manually validated clones by different individuals might be biased by their subjective emotions. 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. We generated a large number of true positive clones with all the different kinds of modifications of the original source codes that possibly generate code clones. We used Mutation Framework [30] for creating such code clone benchmark. The framework takes a code fragment as input, performs mutation operation by random edit operations on the code fragments to artificially create a clone pair. We used 9 different mutation edit operations on the source codes as listed in Table III. 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 biasness by the individuals. We used different original code fragments from BigCloneBench [25] 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 as described in Section III-A. 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 IV. The possible reasoning for this is though the artificially created clones are void of subjective biasness, they have one disadvantage is that they are very similar with one another and comparatively easily distinguishable (as also noticeable from higher recall value in Table IV).

B. Evaluation on different software systems

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-A. 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 CDTs for detecting the code clones available in those open source software project. We used 6 different CDTs to test the generality of the proposed method. Table V lists the CDTs used along with their corresponding version number and configurations. The reported code clones from CDTs were then manually validated by different users. None of the users were previously involved in building the training dataset as discussed in Section III-A. Besides, to compare the performance of the proposed method with similar existing method - FICA, 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 11 shows 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. For 'Java File Manager', however unlike the other systems the proposed approach showed a noticeably lower performance. We manually investigated the system later. We found the considered clones for the system are mostly Type 1 and Type 2 - which may be a possible reason for such a comparative result for the system.

Besides, to test the result quality, system wise precision and recall were calculated for the approaches. The obtained result has been presented in Table VI. As some of the values have been highlighted in the table, it is noticeable that in most of the cases the precision and recall values get lower in comparison to the proposed approach. The result is also

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

TABLE III: List of Operations used to Create Artificial Code Clones via Mutation Framework [30]

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 IV: Result on Artificial Code Clones

Accuracy	Precision	Recall	F1-Score
90%	0.89	0.99	0.93

TABLE V: Used Clone Detection Tools for the Study

CDT	Ver.	Tool Configuration
iClones [28]	0.2	mintokens = 50, minblock = 20
NiCad [13]	4.0	blocks, 30%, 6-2500 lines, blind-renaming, abstract-literal
SimCad [31]	2.2	generous, 6+ lines, blocks
CloneWorks [32]	0.2	Type-3 Aggressive, 6 lines, blocks
Simian [33]	2.4	6 lines, ignore overlapping blocks, balances parentheses
Ctcompare [34]	3.2	50 tokens, 3 replacements

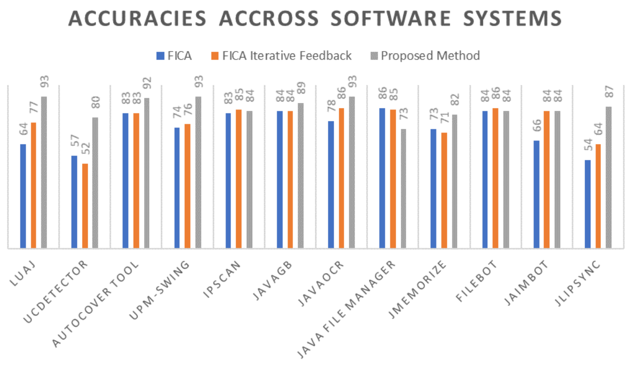


Fig. 11: Accuracy Comparison of the Methods Across Different Software Systems.

TABLE VI: 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 V)
[2] Average per code clone fragment

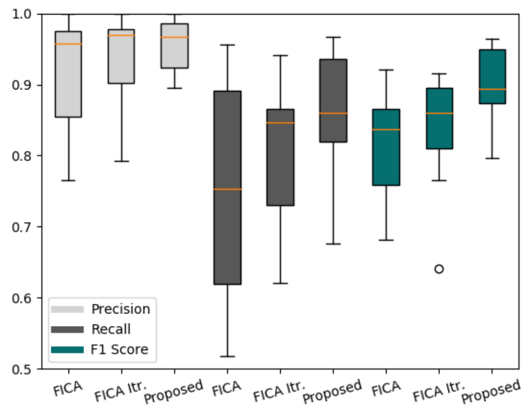


Fig. 12: Result Quality Comparison of the Methods

noticeable in box plot in Figure 12. 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 relatively better and consistent result with lesser variation in the result qualities. Another observation is that, as FICA learns by token sequence comparison, it gets significantly slower as the overall size or the total number of tokens increases for a system. For example, considered lines of code for 'Luaj', were 36155 with an average of 79 tokens per clone fragment, resulting the classification to take noticeably longer time than the proposed approach. We got the same behaviour for similar relatively bigger software system like: 'Java File Manager', 'Upm-swing'and so on.

C. 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 five 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 five clone detection tools (as mentioned above) from five different open source software systems (as mentioned above). 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.

VI. Result Discussion

ANNs are efficient computing models which can approximate any complex functions that has been widely used for pattern recognition in different branches of computer science [35], [36], [37]. On the other hand, one of the major criticisms is their being black boxes, since no satisfactory explanation of their behaviour has been offered. That is ANNs are only given the inputs in the input layer and informed about expected output from the output layer. ANNs then assign required node biases and layer connection weights to predict accordingly without providing us much information about the complex function it learned or how it learned. So from the perspective of our proposed method also, it is challenging to know the nature of the function the Neural Network has learned or if it is giving its decision biasing completely on any of the features used. However, assuming the Neural Network as black box in the middle of input sets and its predicted decision we tried to find out if there is any biasness on any feature of the Neural Network on its 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 13, shows the scores of some of the selected features having higher scores out of all possible extracted

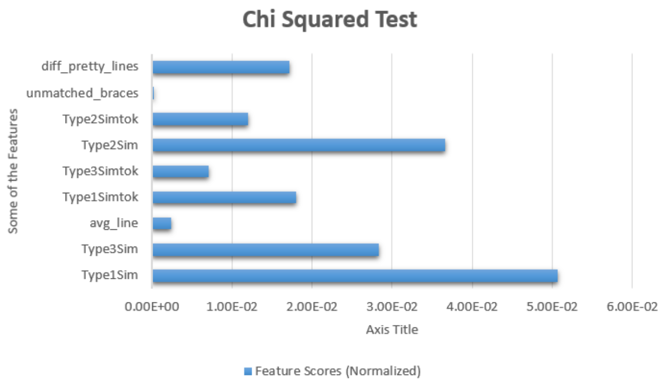
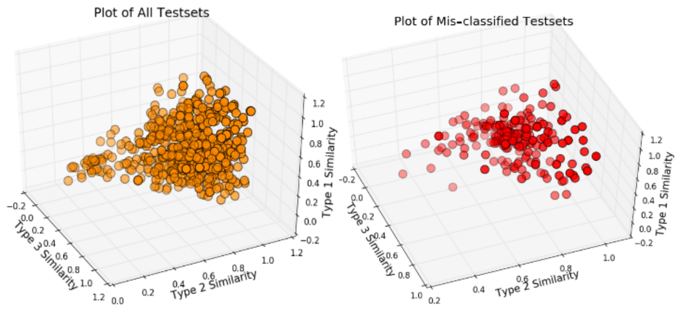


Fig. 13: Feature Score via Chi-Square Test

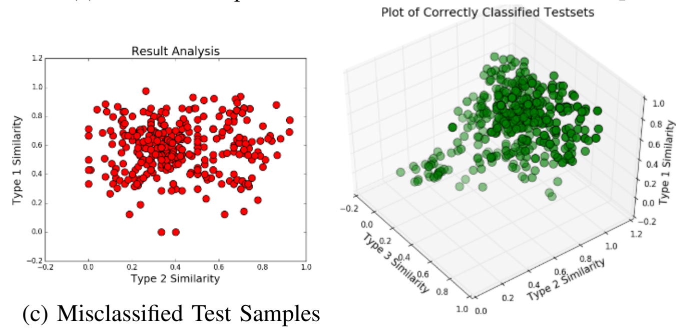
features. From the figure we can see that the normalized score is kind of randomly distributed over the features rather than being completely dominated by one or more features. This gives us some idea that the trained model is not noticeably biased on any feature(s) on its decision making. Besides the 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.

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 or there can also be possibility that the proposed method fails to validate all the Type 1 clones and so on. Especially Type 3 clone is different and difficult to validate in comparison to Type 1 or Type 2 code clones. That is depending on the given type of code clone there are some difference in the validation processes. This leads to the possibility that any proposed method may work only with some particular type(s) of clone(s). 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 result is shown in Figure 14. The top left plot of the figure 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. The top right plot of the figure 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 clone. 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. Besides the bottom left plot of the figure shows a single plane (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



(a) All Test Samples

(b) Misclassified Test Samples



(c) Misclassified Test Samples (2D view)

(d) Correctly Classified Test Samples

Fig. 14: Classification Result Analysis for Different Types of Code Clones

from the randomness of the correctly classified test samples in bottom right plot of the figure.

Based on these findings we answer the research questions as follows:

Answering RQ 1, (Can the manual code clone validation process be assisted via machine learning?): The proposed machine learning algorithm was trained and tested via 10-fold cross validation on a larger dataset (as discussed in Section 3.1). The Neural Network converged within a range of 500 to 600 epochs. Validation accuracy given by the proposed method is found be 87.4% after averaging for each of the 10 folds testing. Besides the trained system was tested by completely different several software systems. We found the proposed method to come up with promising accuracies for clone validation as shown in detail in Table 1. These positive results reveal a lot of opportunities of using machine learning for clone validation to assist in overall code clone maintenance and analysis process.

Answering RQ 2, (Can the machine learning technique predict a programmers validation behavior to improve itself over time and experiences?): The trained model was updated based on the programmers feedback on the clone validation by the proposed method. Each of the feedback are then used by the trained model to update or improve its prediction rule. This way the more a programmer uses this validation method the more it converges towards the validation preferences of the programmer. The same rule also applies for any specific system. This improvement of the validation method was noticeable during its evaluation study with different users and software systems. The proposed method was used for validating 5 different open source software systems as appear in order in Table 1 for each of the users (programmers). That is Filebot software system was validated at first, then Project Libre and so on. For user 1 the validation accuracies (as shown in Table 4) for the consecutive software systems are 55%, 70%, 85%, 85% and 95% respectively. Besides averaging the accuracies for all the consecutive software systems across different users also shows gradual improvement (i.e. accuracies are 58%, 77%, 77%, 84% and 87% respectively). Possible explanation of this gradual improvement is the prediction rule update of the Neural Network based on users feedback in each step. The generalized trained model gradually converges towards the users validation behaviour as more feedback is given to the Neural Network for its weight update to predict validation more accurately. These results indicate the improvement of the method over time learning the validation preferences of specific users and software systems.

Answering RQ 3, (Does the proposed machine learning based validation method works across different clone types and clone detection tools?): To test the generality of validation, the proposed machine learning method was tested with different clone types and clone detection tools. From the evaluation study we found the method does not succeed or fail for any particular type of clones. From the plotting of correctly classified or validated clones by the proposed method in Figure 12, it is noticeable that the clones are randomly scattered across three axes representing the validation works for all the three different types of clones (as previously discussed). Similarly the plotting of misclassified clones also shows randomness across three different axes. That indicates that the clone validation by the proposed method does not succeed or fail for any specific types of clones (for example it might succeed or fail to validate any Type 3 clones or so on). These results demonstrates the generality of the machine learning approach for working across different types of clones. Besides the method was evaluated on validating the clone detection result by 5 different tools. As shown in Figure 10, the validation result in conjunction with different tools found to be quite promising.

VII. Threats to Validity

Neural Networks are widely used for modeling complex non- linear relationships which traditional statistical methods often fail to model accurately. However to learn such complex non- linear functions Neural Networks need a larger training set and also a good amount of time. So if this training phase is carried out by individual programmers, the Neural Network might lack the enough amount of dataset as it needs those clones to have manually validated beforehand for training. Besides even if the individual programmers manage to have such enough amount of manually validated clones, the training process of the Neural Network might take significant amount of time which might reduce the usability of the automatic clone validation. To make this training phase easier we validated a larger set of dataset by 5 different programmers and used them for training the Neural Network. Though it removes the time consuming step of training phase for the individual programmers, it might also raise some threats to validity as the trained model is not based on the individual programmers choice at the beginning. However as the model training was generalized by 5 different programmers independent decision, this possible threats of validity might be considered as minor. Besides to mitigate this possible threat, the Neural Network weights are updated by individual programmers feedback while usages. This way the Neural Network converges towards the validation preferences of the individual programmer while usage over time.

The accuracy and precision of our work across different software systems and clone detection tools were evaluated against pre-judged true positive or false positive clones. These judges can be affected by the subjective preferences on clones of the individual programmers thus raising some possibility of threats to the validation of the work. However we tried to mitigate this possible threats to validity by taking validation decision from multiple programmers.

Another likely threats to validity is possibility of having some minor errors with feature extraction. For the extraction of used features by the proposed method we had to use some source code parsers that works via different transformations of the source code. As the parsers are not always guaranteed to be 100% perfect, the error (if any) might possibly propagate to the feature calculation. However best efforts were given to reduce the probability having any of such minor errors in feature calculations to make the evaluation as accurate as possible.

VIII. Conclusion and Future Work

In this paper we introduced a machine learning approach for automatic code clone validation. Code Clone Detection tools usually return the list of possible clones following some complex searching procedure. The result often contains a number of false positive clone and often does not consider the preferences of users opinion or requirement. This leads to manual validation of the result from the clone detection tools which even gets worse for large-scale software system. We have proposed a machine learning approach that assists automatic validation of the code clones. The method takes feedback from the user to improve its prediction on validation. We evaluated the proposed system with different users, clone detection tools, artificially created code clones and open source project. We found promising accuracy on automatic validation of clones by the proposed method.

While manually validating some code clones by ourselves and also surveying other programmers opinions we found that human judges in addition to finding out structural or syntactical similarity also tries to focus on understanding the contexts or functional similarity of the codes to decide if they are true positive or false positive clones. So our future work plan is finding out some of such features to further improve the result.

References

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

[2] Manishankar Mondal. *On the stability of software clones: A genealogy-based empirical study*. PhD thesis, University of Saskatchewan Saskatoon, 2013.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] Yoshiki Higo, Yasushi Ueda, Toshihro Kamiya, Shinji Kusumoto, and Katsuro Inoue. On software maintenance process improvement based on code clone analysis. *Product Focused Software Process Improvement*, pages 185–197, 2002.

[9] 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.

[10] Brenda S Baker. A program for identifying duplicated code. *Computing Science and Statistics*, pages 49–49, 1993.

[11] Ekwa Duala-Ekoko and Martin P Robillard. Tracking code clones in evolving software. In *Proceedings of the 29th international conference on Software Engineering*, pages 158–167. IEEE Computer Society, 2007.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] 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.

[21] 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.

[22] 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.

[23] 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.

[24] 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.

[25] 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.

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

[27] 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.

[28] 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.

[29] 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.

[30] 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.

[31] 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.

[32] 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.

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

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

[35] Sung-Bae Cho. Neural-network classifiers for recognizing totally unconstrained handwritten numerals. *IEEE Transactions on Neural Networks*, 8(1):43–53, 1997.

[36] Carl Grant Looney. *Pattern recognition using neural networks: theory and algorithms for engineers and scientists*. Oxford University Press, Inc., 1997.

[37] Priscila Lima Rocha and Washington Luis Santos Silva. Artificial neural networks used for pattern recognition of speech signal based on dct parametric models of low order. In *Industrial Informatics (INDIN), 2016 IEEE 14th International Conference on*, pages 46–51. IEEE, 2016.