The Journal of Systems & Software 169 (2020) 110686

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

A machine learning based framework for code clone validation

Golam Mostaeen [a], Banani Roy [a], Chanchal K. Roy [a,*], Kevin Schneider [a], Jeffrey Svajlenko [b]

[a] University of Saskatchewan, Saskatoon, Canada
[b] GitHub Inc.

ARTICLE INFO

Article history:
Received 1 April 2019
Received in revised form 17 April 2020
Accepted 1 June 2020
Available online 22 June 2020

Keywords:
Code clones
Validation
Machine learning
Clone management

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, 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 or irrelevant clones from specific project management or user perspective. To detect all possible similar source code patterns in general, the clone detection tools work on the syntax level while lacking user-specific preferences. This often means the clones must be manually inspected before analysis in order 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. First, a training dataset is built by taking code clones from several clone detection tools for different subject systems and then manually validating those clones. Second, several features are extracted from those clones to train the machine learning model by the proposed approach. The trained algorithm is then 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, automatically evaluate the precision of any clone detectors for any given set of datasets, evaluate existing clone benchmark datasets, or even be used to build new clone benchmarks and datasets with minimum effort. 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 also shows better results in several comparative studies with the existing related approaches for clone classification.

1. 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 (Roy and Cordy, 2007). Such copy/paste practice results in similar pieces of code fragments in a system, called code clones. Researchers agree upon four primary clone types (Roy and Cordy, 2007): 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., additions of new statements, or deletions and modifications of existing ones). Type-4 clones are syntactically dissimilar code fragments that implement the same or similar functionality. Some of the recent research shows that on average around 7% to 23% of total code of a software system is duplicated or cloned from one location to another (Baker, 1995; Kapser and Godfrey, 2006b; Mondal et al., 2011). Although code cloning is often done intentionally to accelerate the development process and also not all code clones are harmful (Kapser and Godfrey, 2006a), 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 (Juergens et al., 2009; Göde and Koschke, 2011; Bettenburg et al., 2009; Saha et al., 2013). For example, it becomes very difficult to carry out consistent changes to all the cloned code fragments throughout the software system. These inconsistent changes to the corresponding duplicated code fragments are often responsible for the creation of new software bugs (Mondal et al., 2017b). In addition to the 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 (Mondal et al., 2017a; Juergens et al., 2009). Detection of such code clones can, therefore, accelerate the maintenance tasks of any software systems remarkably (Juergens et al., 2009). Besides, exploiting the similarities of the detected code clones also helps one better understand and improve the

* Corresponding author.
E-mail addresses: golam.mostaeen@usask.ca (G. Mostaeen), banani.roy@usask.ca (B. Roy), chanchal.roy@usask.ca (C.K. Roy), kevin.schneider@usask.ca (K. Schneider), jeff.svajlenko@usask.ca (J. Svajlenko).

https://doi.org/10.1016/j.jss.2020.110686
0164-1212/

overall software design (Higo et al., 2002; Mondal et al., 2014a,b; Zibran and Roy, 2012a).

At least 70 Clone Detection Tools and techniques have been proposed and developed to automate the clone detection process, for the extensive research in this specific area over the last decade (Kamiya et al., 2002; Baker, 1993; Duala-Ekoko and Robillard, 2007; Bellon et al., 2007; Roy and Cordy, 2008; Jiang et al., 2007; Tairas and Gray, 2006). These tools return a list of possible code clone pairs or classes available in a given software system. Except for Type 1, the other types of code clones (Type 2, 3 and 4) undergo different changes over time and can get too complicated to be detected with a simple string matching algorithm by a clone detection tool. 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 time make the searching problem much more complicated. In order to handle those complex source code structures while still detecting all possible code clone pairs, the tools undergo a lot of generalization of the original source codes like pretty-printing (Roy and Cordy, 2008), normalization of the identifiers (Kamiya et al., 2002; Roy and Cordy, 2008), forming syntax tree (Koschke et al., 2006) of the code fragments and so on just to name a few.

As a result of this complex searching problem and necessary generalization or normalization of the source code, the clone detection algorithms often report false positive clones. These are pairs of code fragments that are not similar or possibly are only coincidentally similar or are otherwise considered not a valid clone by the user (Jiang and Hassan, 2007). 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 (Keivanloo et al., 2015; Charpentier et al., 2015; Yang et al., 2015). For example, Yang et al. (2015) conducted a survey where several users were provided the same clone sets for validation detected by clone detection tools. The study reported significant variations among the users in validating the same clone sets (e.g. for the same provided clone sets, the number of decided true positive code clones varied within a range of 4.76% to 23.81% for different users). For these reasons, programmers often need to manually validate if the results of a clone detection are a true clone or not before using this information for the given specific scenarios like source code refactoring or other software maintenance tasks. Such a manual validation process becomes a hindering factor even for a medium sized software system. 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 (Jiang and Hassan, 2007; Tairas and Gray, 2009). 15% of the total lines of code of the Linux kernel has been reported as code clone which is 122 K LoC (Li et al., 2006). Both of the above scenarios on the 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. Thus, in the best case even if a tool returns only true positive clones, many of those clones might not be relevant to the tasks at hand of the programmers or engineers (e.g., not suitable for refactoring) (Jiang and Hassan, 2007). Mining those code clones of interest from the tool generated report is often a time-consuming task and thus reduces the usability of code clone detection tools. The scenario gets even worse with the increase of software project in size. Studies also show the subjective nature of detected clones requiring manual validation before usage. For example, in the conducted user study of Yang et al. (2015) involving 105 detected clone pairs and multiple expert judges were asked for manual validation. Among the set of 105 detected clone pairs, the independent manual judges showed significant differences in the validation patterns (i.e., as presented above). For example, one of the user judged 5 of the clone pairs as false clone, while other users judged 24, 23 and 25 of the clone pairs as false clones respectively.

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 the time-consuming manual validation process. We studied performance and result qualities of different machine learning algorithms in validating the detected clones. We also extend the proposed method with a cloud-based architecture to ensure the compatibility of the proposed method with any of the existing clone detection tools. We got promising results from our several studies with different experimental setups for the clone validation. The proposed method also showed the better result in a comparison study with related existing works for code clone validation.

This study is aimed to answer the following 2 research questions:

- **RQ 1**: Can the manual code clone validation process be assisted via machine learning?
- **RQ 2**: Does the proposed machine-learning based validation method work across different clone types and clone detection tools?

Our work makes three main contributions. *First*, we propose a cloud-based architecture for automatic code clone validation using machine learning. We also present an implementation of the working version of the tool as a proof of concept of the proposed method. The open-source tool is available at GitHub[1] for further extension and contribution to the research domain. *Second*, 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 result improvement in this research area. *Third*, we conducted a detailed comparative study with 11 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 the strengths and weaknesses of several machine learning algorithms on clone classification can contribute to future research in this area for further improvement of the learning model.

The rest of this paper is organized as follows: Section 2 provides a discussion of related works on this specific research area. Section 3, contains a discussion about the proposed method. We then present the data distribution analysis and comparative analysis of multiple machine learning algorithms in Section 4 and Section 5 respectively. Section 6 discusses the several experiments we conducted with proposed and existing related methods. The result discussion and possible threats to validity are presented

[1] https://github.com/pseudoPixels/CloneCognition.

in Section 7 and Section 8 respectively. Finally, in Section 9, we discuss our conclusion and future work.

2. Related works

In this section, we introduce existing research works which targeted the clone classification problem and closely related to our proposed method for automatic clone validation.

Though several methods and techniques have been proposed over the past few years for maintenance, organization or classification of code clones, very few of them recently focused on aiding the huge manual user-specific validation task of the reported code clones. Yang et al. (2015) studied the similar problem for user-specific code clone classification in their work — FICA. The user-specific clone classification in FICA is done by token sequence similarity analysis using 'Term-Frequency–Inverse Document Frequency'- (TF-IDF) vector. For training some of the reported code clone pairs from clone detection tools are manually marked as True or False positive clones by the users. So, the whole training set M, is divided into two sets of clones- M_t and M_f such that $M_t \cap M_f = \emptyset$ and $M_t \cup M_f = M$. Tokens are then extracted from both the clone sets to produce an 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).

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

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

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

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

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 as in Eq. (4).

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

Here, $w(m)$ is the assigned weight in the range [0, 1], in the case of numerical range marking of the clones. However, in case the of 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 the 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.

Saini et al. (2019) recently proposed an approach and a tool (*InspectorClone*) aiming to semi-automate the precision studies of clone detection tools. Since their work is focused on clone precision reporting in contrast to our validation framework, the workflow requires human judges in the loop in order to provide a final precision report. For the precision reporting, InspectorClone tries to validate only a subset (i.e., the clone subset that it can validate with high confidence and precision score) of given clone pairs and the remaining of the clone pairs (i.e., which it could not resolve with high precision) are given to human judges for manual validation. Finally, the tool then merges the two clone subsets to aggregate and return a final precision evaluation of the target clone detection tool. The human involvement (i.e., as they refer their approach as semiautomated) is reasonable as the tool is responsible for providing the final precision score of the target clone detection tool and it needs to be done with very high precision, otherwise the whole precision reporting of a target clone detection tool will not be reliable. However, in contrast, our proposed validation framework is targeted for assisting users in the overall analysis of the detected clones and fully automated based on a pre-trained model. As we envision our proposed machine learning based approach to improve over time with an increased number of datasets and learned human validation patterns, we plan to explore the possibility of even fully automating the precision reporting process of the clone detection tools.

In recent years a number of research studies have been done for code clone detection tools' reported clone classification or comprehension. For example, Tairas and Gray (2009) 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 (Kapser and Godfrey, 2004), type similarities (all possible Type 1 clones grouped together and so on for Type 2, 3 and 4) of the detected clones (Bellon et al., 2007), and Latent Semantic Indexing (LSI) on the identifiers of clones (Tairas and Gray, 2009). 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 (Yang et al., 2015), applying unsupervised machine learning algorithm to create clone clusters (Svajlenko and Roy, 2016). On the other hand, the second category of clone comprehension techniques works based on their visual representation: scatter plot of the code clones (Kamiya et al., 2002), an aspect browser-like view (Tairas and Gray, 2009), and hierarchical graphs of detected clones (Jiang et al., 2007). These classification and visualization techniques can make the organization and maintenance of code clones are much easier. We can notice that the total number of clones to be manually analyzed for validation still remains the same. The overall result of such code clone comprehension techniques can be improved significantly by adding an automatic validation process that uses a 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 the validity due to unavoidable human errors and need a huge amount of manual validation work. For example, Bellon et al. (2007) created one such benchmark by validating 2% of the union of six clone detectors for eight subject systems that required 77 h of manual efforts, Svajlenko et al. (2014) 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 the 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 behavior of the specific system or programmer. The proposed method extracts 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 extracts 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 programmer's 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 programmer's preferences even better with time and experience. Besides, the proposed method can optionally be tuned to control the validated result based on the importance of the code clone pairs including in the IDE-based management of clones (Zibran and Roy, 2011) or when searching clones in the IDE (Zibran and Roy, 2012b). 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 flexibility on selecting the number of validated clones for given scenarios.

3. Proposed approach

In this section, we discuss the proposed method for the clone classification problem. In Section 3.1, we first present the high-level workflow of the proposed method for training the machine learning model. Section 3.2 to Section 3.5, contain the discussions detailing some of the significant workflow steps. And finally, we present the extended cloud-based architecture of the proposed method in Section 3.6.

3.1. Overall workflow of the proposed method

Fig. 1, shows a high-level workflow (which has been extended for the cloud-based architecture as discussed in Section 3.6 in details) of the proposed method. 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. The workflow steps can be listed in sequence like the following:

1. In this step, source code from Codebase are supplied to any of the existing clone detection tools.
2. The detected code clones from the corresponding clone detection tools are collected in Step 2. As the proposed method works with any of the existing clone detection tools, clones from multiple tools can be combined optionally for further generality of the training set.
3. The reported code clones from clone detection tools are provided to the user for manual validation.
4. User marks the code clones as true positive or false positive in Step 4 (details in Section 3.2). The user-specific manual clone validation results are stored in a database for use as a training set of machine learning models.
5. Several features are extracted from the marked code clone pairs for training the machine learning models. The existing related research works (for example FICA (Yang et al., 2015)), used only simple token sequences as features for training the machine learning model and thus failed to predict the validation successfully beyond Type 2 clones. To improve the classification results and to target clones beyond Type 2, we considered calculating clone similarity with several levels of structural pre-processing and normalization. The details of the feature extraction methodologies have been presented in Section 3.2.
6. The extracted features are used to build the feature vector for clone classification. Feature vectors for the corresponding manual validation clone classes are used for training machine learning model in Step 6.
7. Next, in Step 7, the trained machine learning model is used for predicting the clone validation pattern for the unknown or test sets. The machine learning model at this stage returns the probability score (of being true or false positive) for the given corresponding code clone pairs.
8. Finally, in this step, the classified result is sent back to the clone detection tools. The classification result can be tuned based on user preference (of probability score) for the final result. The system can take user feedback based on the classified clones from the user for repeating the cycle of supervised learning, thus improving the validation result over time.

3.2. Clone labeling and feature extraction

The reported code clones from the clone detection tools are provided to the user for manual validation (Step 3, Fig. 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 a 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.

As machine learning models learn to map the input feature sets to the corresponding class label, it is important to select appropriate features for the given classification problem. For example, Yang et al. (2015) targeted Type 2 clones in a similar study of the code clone classification problem and hence used simple token sequences as features for using in the classification algorithm. As we intended to enhance the classification performance (so that it works efficiently with more diverse types of clones and also shows better prediction score), in addition to improving the whole classification workflow, we also focused on extracting more informative features. Most of the selected feature extraction undergoes two main steps: (i) Pre-processing and source code transformation and (ii) Similarity extraction from the code clone pairs.

```
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: Sample Code Clone (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: Sample Code Clone (Fragment 2)

Pre-processing, like pretty-printing and comment removal ensures consistent structures for matching and similarity extraction of similar source code pairs (for example Type 1 clones). Extracting similarity features between the two code clone fragments after this step (i.e. comment removal followed by pretty-printing) gives us the information about how a user sees the code clones for validation. Thus at this point, the extracted features represent mainly Type 1 similarity between the target code clone

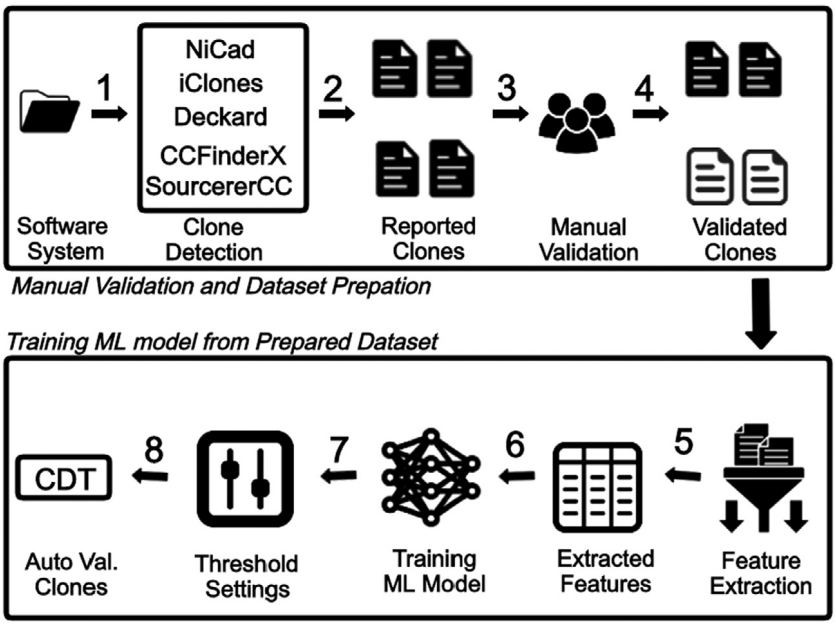

Fig. 1. Workflow of the proposed method.

fragments. In addition to that, different source transformations like consistent normalization of literals or consistent renaming of identifiers are applied to consider the possible changes between the code clone pairs (i.e. for Type 2 and Type 3 clones). For example, Listing 1 and Listing 2, show the code fragments of one of the detected clones from *Weka* software system (Hall et al. (2009)), that needs to be validated (comparatively simpler and straight forward code clone pair used for discussion).

Though the code clone pairs exhibit much structural similarity, calculating similarity directly based on original source code pairs have higher probability of introducing noise (from the perspective of Type 2 or Type 3 clones) due to strict consideration of the modifications of literals and identifiers. So, we also applied different source code transformations before calculating clone similarity for extracting possible Type 2 or Type 3 information. For example,

```
try {
    if (X.X == 0) {
throw new X(
    "string");
    }
    X X = X[0];
    X[0] = "string";
    X.X.X(X(X, X));
}
```

Listing 3: Pre-processed and Transformed Code Clone (Fragment 1)

```
try {
    if (X.X == 0) {
throw new X(
"string"
+ "string");
    }
    X[0] = "string";
    X X = X.X(X, null);
    X.X.X(X(X, X));
}
```

Listing 4: Pre-processed and Transformed Code Clone (Fragment 2)

Listing 3 and Listing 4, show the transformed clone fragments from Listing 1 and Listing 2 respectively, after the first blind renaming of identifiers and then applying consistent normalization of literals. For example, after the blind renaming of the identifiers, all different identifiers take a common name, X for the structural comparison. Similarly, all the different string literals were transformed to - "string" after consistent normalization of the literals as shown in Listing 3 and Listing 4. These transformations allow the corresponding modifications of literals and identifiers and thus provides similarity feature information for Type 2 and Type 3 code clones. We used TXL (Cordy et al., 1991) for different source transformations.

After applying different pre-processing and transformation for different types of features, we then analyze the differences between the code clone fragments (i.e. the output of the previous steps). Prior to calculating numerical similarity values between the clone fragments, we find out the minimal changes required to transform from one clone fragment to another. For example, Listing

```
4c4,5
<     "string");
---
> "string"
> + "string");
6d6
<     X X = X[0];
7a8
>     X X = X.X(X, null);
10,12d10
<
<
<
```

Listing 5: Difference between the code clone fragments

5, shows the minimal changes or operations required for transforming code clone fragment 1 (i.e. Listing 3) to code clone fragment 2 (i.e. Listing 4). We used the *Unix Diff* utility for the purpose, that calculates the minimum set of insert and delete operations required for converting one file to another. For example, in Listing 5, $<$ and $>$ signs — represent delete operation, $\mathcal{O}_d$ and insert operation $\mathcal{O}_i$ respectively, that we need to apply on first clone fragment for the required transformation. For example, the first conflict of the transformed clone fragments in Listing 3 with Listing 4 is at line 4. The minimum operations needed to resolve the difference is one delete operation, $\mathcal{O}_d$ of the original line at 4,

followed by two insert operations, $\mathcal{O}_i$ of line 4 and 5 from Listing 4. The corresponding change operations have been represented as $4c4, 5$ in Listing 5. We then calculate the similarity value between the two code clone fragments f_1 and f_2 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 count of the corresponding change operation and length of the corresponding code clone fragment respectively. The fragment similarity thus falls in the range of [0,1]. As the number of such differences between the two code clone fragments increases, the code clone fragments similarity measure tends towards zero. On the other hand, the fragment similarity is calculated as 1, in case the clone fragments are exactly similar with no further required changes (i.e. $C(\mathcal{O}_d) = C(\mathcal{O}_i) = 0$).

We also used several other features to get more structural information about the two code clone fragments. We tried to mimic several manual validation patterns as per our obtained experiences on manual code clone validation of users. For example, our intuition was if the code clone fragments are significantly different in size, a validator may be more likely to mark them as false positive. The corresponding code clone fragment sizes α and β, were calculated as respectively. The difference $|\alpha - \beta|$, provides information about the variation of fragment sizes. Smaller difference values represents more likelihood of being validated as true positive code clone than that of comparatively higher difference values and thus was considered as one possible feature for the clone classification problem. However, for a clone that is small versus a clone that is large might have different consideration. For example, for a relatively larger code clone fragment pair, it is possible to have more variance in difference than that of smaller code clone fragments pair. So, to mitigate this possible bias we also considered the average size of the code clones $(\alpha + \beta)/2$. That average value captures the size of the clones and difference captures if the code fragments are rather mismatched in size.

Please note that some of the popular code clone detection tools use source transformations like consistent renaming of identifiers, or normalization of literals, as part of their workflow for code clone detection. A few such clone detection tools like NiCad (Roy and Cordy, 2008) and CCFinder (Kamiya et al., 2002) are also well known in the research area for their performance on clone detection. So, with this motivation, for a subset of the features we also carried out similar transformations before calculating the clone fragment similarities as discussed above. However, to the best of our knowledge no previous works on clone validation used a 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. For a given feature from the feature set, we tried to find out its class separability for the two classes and its overall contribution score for the classification. Some of the features showed higher contribution score while a few a of them showed comparatively low contribution score or class separability. The feature study provided us the information about higher contributing features while removing a few of them that more or less exhibit as possible noise for the clone classification problem. Section 4, discusses our findings in details about the feature sets for the classification of the code clones.

3.3. Training machine learning models for clone classification

As we have presented the workflow of the proposed method in the above discussion, it uses a supervised machine learning algorithm for learning the classification pattern of the user-specific clone validation (i.e. in Step 6). The supervised classification algorithm will be trained on the manually validated datatset $D = \{(\mathbf{x}_1, \mathbf{y}_1), (\mathbf{x}_2, \mathbf{y}_2)...(\mathbf{x}_m, \mathbf{y}_m)\}$, 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 D, 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 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 sequence matching algorithms instead (for example, TF-IDF, token sequence matching) and failed to validate beyond Type 2 clones (for example FICA (Yang et al., 2015)). We studied the performance of multiple machine learning classification algorithms, for example, Random Forest, Naive Bayes Classifier, Decision Table, Backpropagation Neural Networks, Deep Learning. In the comparative study of such 13 different machine learning algorithms, we got accuracies within a range of 76% to 87% for clone classification. Backpropagation Neural Network resulted in maximum performance in comparison to other classification models with an accuracy of 87.4%. So, we used the Backpropagation Neural Network as the machine learning model for the experiments of the proposed method (as discussed in Section 6). Our findings and results of the comparative study of multiple machine learning classification algorithms have been discussed in detail in Section 5.

3.4. Prediction decision configuration

The machine learning models classify the test code clone pairs using the extracted feature vector $\mathbf{x}_t$. Probabilistic classifiers learn a function f, such that $f(\mathbf{x_t})$, assigns probability 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. In the proposed method, users can set the decision threshold $\gamma[0, 1]$ to tune the validation output quality. 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 proposed method becomes more strict for classifying clones and will return only those clone pairs having a higher probability of being true positive clones. Thus most of the returned results are expected to be true code clone pairs. Similarly one can decrease the value of γ to make the proposed method more tolerant for classifying the clones in true positive class. That is this decision threshold can be useful for the users to tune the result quality as per the requirements.

3.5. Improving machine learning model with supervised user feedback

The classified code clone pairs from the trained machine learning model are sent back to the corresponding code clone detection tools. User feedback on those test code clone pairs can be collected and stored in a database by the proposed method. The feedback code clone pairs can be optionally used along with the existing training set to further improve the code clone classification. This cycle of supervised learning adds the possibility of improving the classification accuracy over time and experience.

The pre-trained model can be used in *transfer learning* (Pan and Yang, 2009; Torrey and Shavlik, 2010) towards developing custom validation models for given use-cases with the manual validation feedback from the cycle of supervised learning. The goal of the transfer learning is to improve or customize the target task leveraging the knowledge from source task (Torrey and Shavlik, 2010). Hence, our pre-trained Artificial Neural Network model from large set of training dataset can be used to improve or customize the validation model with comparatively minimal effort.

3.6. 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. Machine learning model generalization and performance significantly depends on the quality and quantity of the training dataset. With relatively more data, the machine learning models are expected to perform better. As the proposed method can work on top of any clone detection tools generated results, to increase the usability we propose a cloud-based architecture. Fig. 2, gives an overview of the architecture. Reported code clones from a target code clone detection tool are sent to the server for validation using an 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 and language of the code clone source code. The communication with the server is done using JavaScript Object Notation (Crockford, 2006) (in key–value pairs as an example shown in Fig. 2). On receiving the subject clones to validate, the required features are extracted to build the 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 on the user end.[2] Some of the advantages of the cloud deployment of the model can be discussed as the following:

3.6.1. 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 (for example the proposed method uses JavaScript Object Notation) that is understandable by all the tools irrespective of their implementation varieties, and can thus improve the usability of the proposed method to a great extent.

Listing 6 shows a sample REST API request for automatic clone validation. As it follows standard REST API request format, any of the existing clone detection tool can make such requests for automatic clone validation irrespective of the underlying architecture or implementation languages of the corresponding tools. Getting the validation requests, the proposed validation tool makes all pre-processing of the clone fragments, uses the pre-trained cloud deployed model for making prediction and finally sends the validation response to the corresponding clone detection tools (i.e., as presented in Listing 7). The architecture thus ensures compatibility with any clone detection tools with minimal efforts.

```
{
    "lang": "Java",
    "sourceCode_1" : "<code clone fragment 1>",
    "sourceCode_2" : "<code clone fragment 2>"
}
```

Listing 6: Sample REST API request for clone validation

```
{
    "output": {
        "prob_false_clone_pair" : 0.1
        "prob_true_clone_pair" : 0.9
    },
    "log_msg" : "Preprocessing clones, Normalizing Codes ,...",
    "error_msg" : None
}
```

Listing 7: Sample REST API response with automatic clone validation

3.6.2. Improvement in the training phase

Getting enough training data or time by an 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 dataset is also simpler and involves a single request to the cloud. Additionally, the user can also choose among different trained models (i.e. Artificial Neural Networks or Decision Tree) for better convergence with their decision. Besides, the cloud-based architecture also adds the possibility of managing a common knowledge base of validation for a user group working on a specific project. For project-specific clone analysis, the project team often targets particular code clones of interest as per the task at hand. In these cases, the cloud-based architecture can be useful for managing the training dataset for common validation patterns.

Cloud-based model also opens up the possibilities for higher processing power with cluster or distributed computing for future works. Thus the higher processing advantages for big data of clone validation is possible with even from relatively low processing power end devices for analysis.

4. Studying data distribution for the clone classification

In the above Section 3, we provided a high-level workflow for the usage of machine for code clone validation. In this section, we present our data distribution study (i.e., in terms of true positive and false positive code clones) for the feature selection of the machine leaning model. We have divided the discussion of this section on dataset description in three parts. In Section 4.1 we discuss the data sources that we used, next Section 4.2 contains the discussion on high-level details of the dataset that was used to train and test the system and finally in Section 4.3 we discuss about some statistical summaries and underlying distribution in terms of different extracted features.

4.1. Data source

As the machine learning algorithm tries to recognize any available underlying pattern in the given dataset, it is 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, resulting in the failure to generalize all the other types of clones. So to get generalization in validating different types of code clones by the system we have chosen to use a relatively bigger and diverse dataset of open source projects. Besides, we also considered clones reported by different existing clone detection tools from those multiple open source projects.

For training, we used clones from IJaDataset 2.0 (Ambient Software Evoluton Group, 2013), - 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 (Roy and Cordy, 2008), Deckard (Jiang et al., 2007), iClones (Göde and Koschke, 2009), CCFinderX (Kamiya et al., 2002) and SourcererCC (Sajnani et al., 2016) were used to detect clones separately out of the benchmark. While we could use a different set of clone detection tools, our target for the generality

[2] We implemented a prototype of the proposed system. All the source codes and required resources will be available after the blind review phase.

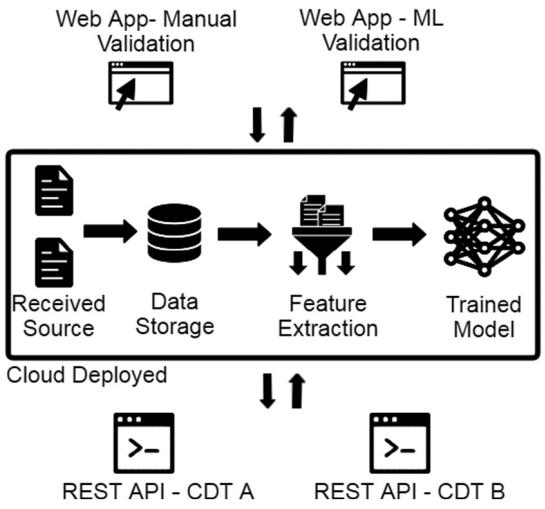

Fig. 2. Cloud model for compatibility with existing code clone detection tools.

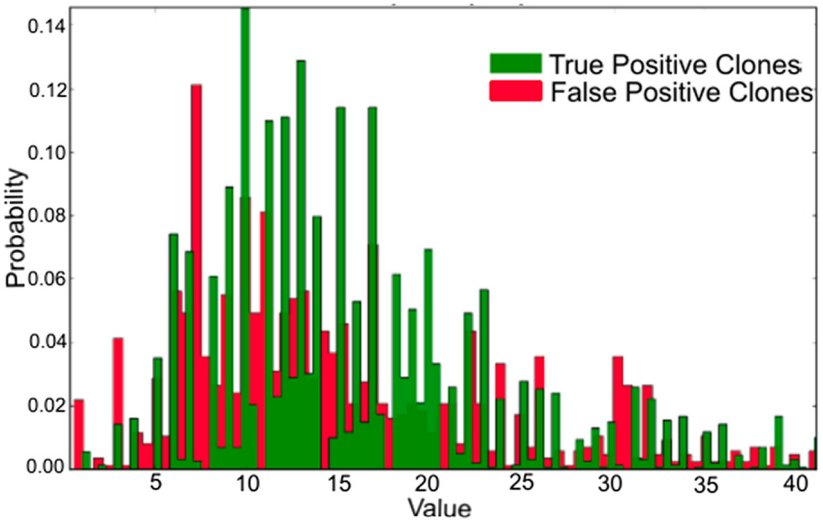

Fig. 3. Histogram of code fragment average.

test suffices by the selected tools. The selected clone detection tools are widely used and researched in recent time, they provide a common ground for the experimental evaluation. In addition to that, as the proposed method works directly on the clone fragments (i.e., for feature extraction), it should be compatible with any clone detection tools, as all of the tools return some meta-data or information about the clone fragments from the original software system.

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 the proposed approach.

4.2. 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 purposes. 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 user's perspective (Keivanloo et al., 2015), 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 consider this generalization to the proposed method the whole set of code pairs were split into five parts to be validated by five different graduate research students from computer science background. This manual validation decision along with the corresponding possible code clone pairs are given as input to the proposed method for the training phase.

4.3. Analyzing data distribution for the clone classification problem

Out of those manually validated clones we extracted different features that are used to train the machine learning model. In this section, we discuss different distribution and statistical studies and behaviors of some of the extracted features.

For every code clone pair detected by clone detection tools, we found the similar code fragments for a clone pair. These are the similar code fragments for which the tools decided could be a code clone pair. We analyzed this feature of the code clone

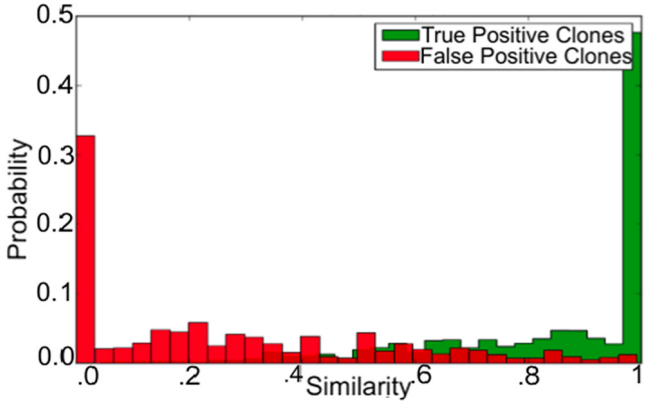

Fig. 4. Histogram of syntactical similarity by line (Type 1 Norm).

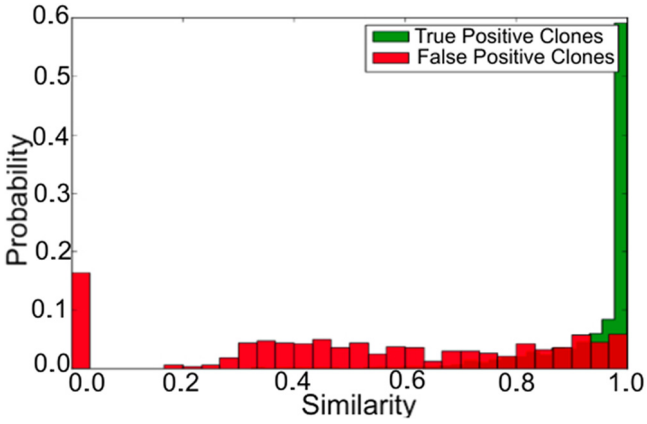

Fig. 5. Histogram of syntactical similarity by token (Type 1 Norm).

pairs for both true positive and false positive manually validated clones in an attempt to find its contribution score for clone classification. Fig. 3, shows the distribution of the average code clone fragment feature $((\alpha + \beta)/2$, as discussed in Section 3.2) 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 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, the syntactical similarity was measured by lines and by tokens for the clone pairs resulting in 6 different possible features (Section 3.2). To view any underlying distribution of the features their normalized histogram were plotted both for true positive and false positive clones. Fig. 4, shows one 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 better possible contribution score for validation prediction than the average clone fragment sizes. Fig. 5, also shows somewhat similar results in the case of Syntactic

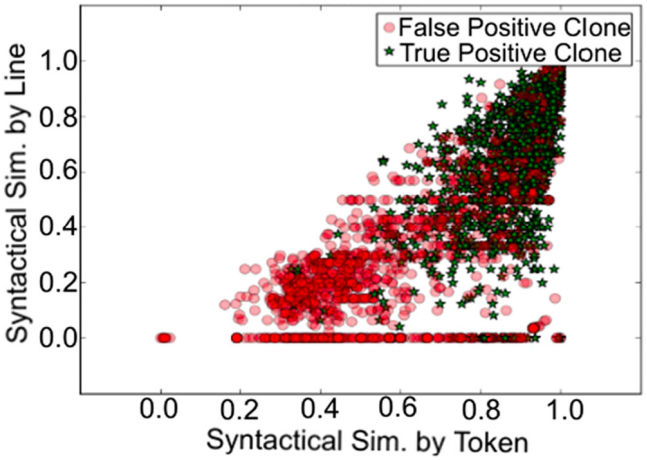

Fig. 6. Syntactical Sim. by line vs token (Type 1 Norm).

Similarity measured by tokens after Type 1 Normalization of the source codes.

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 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 results for visualization in an attempt to notice any distinguishable separation or clusters. For example, Fig. 6, is one 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. However, these analyses did not show any distinguishable cluster information for the two classes.

As machine learning algorithms try to recognize any underlying pattern available on the working dataset, the detailed analysis on the dataset and possible features are necessary for selecting the right machine learning algorithm and features. This distribution analysis on different possible features for code clones provides information about their importance and contribution for clone validation. This analysis provides a clearer view of the data distribution and thus helps to pick the appropriate machine learning algorithm and corresponding features for the algorithm. From several analyses on the data distribution, we tried to find out the features that have comparatively more distinguishable distribution and provides more contribution for the two classes — true positive and false positive clones. Table 1, shows a feature set ranked on possible contribution score based on our analysis study. The corresponding distribution mean differences, $\Delta\mu$ for the two classes also somewhat indicates the separability for the classification.

The detailed feature study, in terms of class distribution prior to applying any machine learning algorithm is very important, since using any noisy feature (for the specific classification problem) may affect the classification performance and reduces the generality of the classification. The distribution study, also contributes to the research area for further improvement in feature extraction and selection of appropriate classification algorithms. From our study, we built the feature vector as listed in Table 1. The other features were not used for the clone classification due to their low contribution scores or noisy behaviors for the classification as discussed above.

5. Performance analysis of different machine learning models: A comparative study

In the last Sections 3 and 4 we presented the high-level workflow involving machine learning and data distribution analysis for

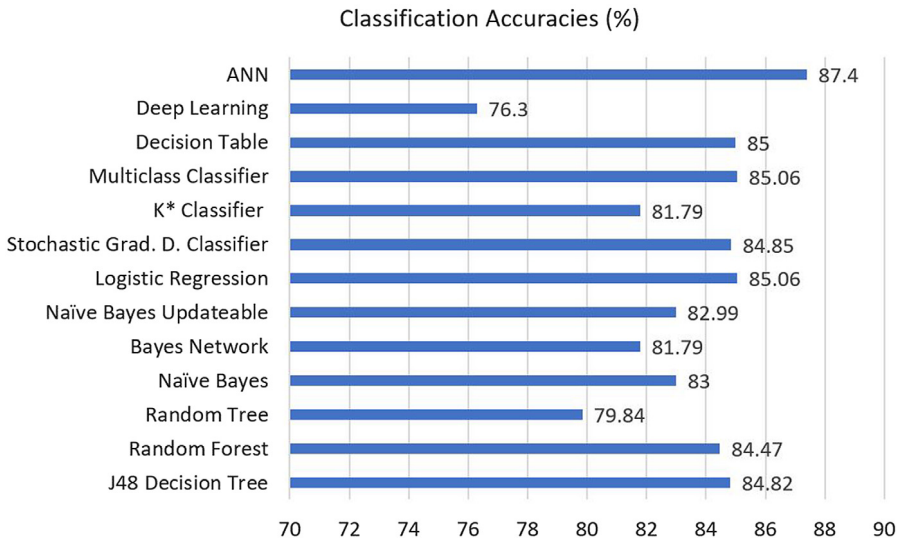

Fig. 7. Accuracy comparison of the methods across different software systems.

Table 1
Selected features based on distribution analysis.

Feature	$\Delta\mu$	Feature summary (as discussed in details in Section 3.2)
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

feature selection by the classification models respectively. In this section, we study the performance of different machine learning models in our proposed high-level workflow with the studied feature set.

5.1. Bayes classifiers

From the extracted code clone feature vector $\mathbf{x} = (x_1, x_2, \ldots, x_n)$, we experimented with the Naive Bayes Classifier - a conditional probability model, for classification of the clonesets into two clone classes - C_T and C_F, representing true and false positive validated clone classes respectively. For the extracted n clone features of a reported clone pair, the classifier assigns conditional probabilities for the two classes - $Pr(C_T|x_1, x_2, \ldots, x_n)$ and $Pr(C_F|x_1, x_2, \ldots, x_n)$ using the Bayes' Theorem as Eq. (5):

$$Pr(C_k|\mathbf{x}) = \frac{Pr(C_k)Pr(\mathbf{x}|C_k)}{Pr(\mathbf{x})} \tag{5}$$

where, $k = \{T, F\}$, $Pr(C_k)$ is the prior probability of the clone class C_k, $Pr(\mathbf{x}|C_k)$ is the likelihood of the clone pair with feature vector $\mathbf{x}$ to be in the clone class C_k and $Pr(\mathbf{x})$ is the evidence of the feature vector $\mathbf{x}$. The evidence can be ignored as it is independent of the clone class C_k. Under the assumption that the clone feature x_i is independent of any other feature x_j for $i \neq j$, Naive Bayes Classifier then assigns the class probabilities for a given test feature vector $\mathbf{x_t}$ as Eq. (6):

$$Pr(C_k|\mathbf{x_t}) = Pr(C_k)\prod_{i=1}^{n} Pr(x_i|C_k) \tag{6}$$

We used kernel density estimation (John and Langley, 1995) for the likelihood calculation as most of the selected feature values are continuous. With described configurations the classifier

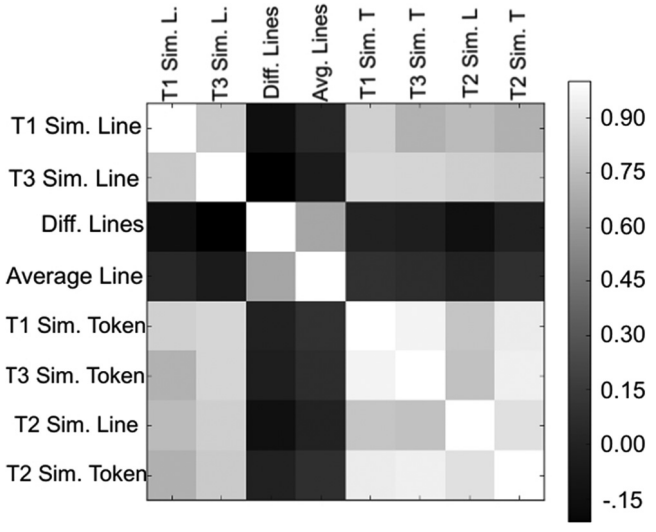

Fig. 8. Correlation among a feature subset.

showed an accuracy of 83%, with 0.831 and 0.830 of precision and recall respectively (More result quality analysis reports in Table 2).

However, it is a strong assumption by the classifier to consider independence among the extracted features for a given class C, for the clone classification problem. Because, by the definition of the code clones (Roy and Cordy, 2007), it is usual that part of a Type 2 clone can contain Type 1 clone. Similarly, Type 3 code clone can also contain fractions of Type 2 or Type 1 clones. So, by induction it is expected that the extracted similarity

features (Section 4.3) for a given clone class have some sort of correlation among them. For example, Fig. 8, shows the correlation among some of the extracted features. As it is noticeable from the figure, the clones structural features such as average line or line differences shows relatively lower correlation with other features. However, the extracted similarity based features among clone pairs after different levels of normalization shows significantly higher correlation among them. From these findings, we experimented with the Bayesian Network Classifier (Friedman et al., 1997) - that considers and learns the possible dependency relations among the features. Unsupervised learning was used (via Minimum Description Length (MDL) (Lam and Bacchus, 1994) scoring method) to build the dependency network.

Though 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 (as shown in Fig. 7). This behavior is not totally unexpected though, as Friedman et al. (1997) showed a detailed study on this. Error while learning the dependency network from the training set was presented as possible reasoning for such behavior.

5.2. Decision tree classifiers

For predicting the target variables, these classifiers build a decision tree from the input variables of the used feature vector $\mathbf{x}$. The internal nodes of the tree correspond to different input variables, values of the corresponding input variables define the edges connecting nodes and each leaf denotes different target variable for the classification. At each step of learning the model, an input variable is selected as a node, such that it best splits the remaining training dataset. There are several variations of the classification based on this recursive training set split scoring and corresponding node selection. We used a number of them for testing the performance for the clone classification problem. The Pruned C4.5 decision tree (Quinlan, 1993) showed an accuracy of 84%. The obtained precision and recall are 0.849 and 0.848 respectively. Random Tree is also another variation of the classification group that considers K random input variables at steps for generating the decision tree. The obtained accuracy was 79%. The precision and recall values were also relatively lower than C4.5 decision tree. We also experimented with Random Forest classifier (Breiman, 2001), that considers multiple tree decisions for building the model. We got approximately similar accuracy with this classification algorithm as C4.5 decision tree. However, precision and recall values show relatively lower values than C4.5 decision tree.

5.3. Backpropagation neural network

From the training dataset D, 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, the ANN learns the following function:

$$f(\mathbf{x}) = \sigma(W_{ho}^{\tau} \cdot \sigma(W_{ih}^{\tau} \cdot \mathbf{x} + \theta_h) + \theta_0) \quad (7)$$

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 denote the layer bias and neuron activation function respectively. We used softmax activation function for the output layer. The learned function $f(\mathbf{x})$, is then used to predict the clone validation for the new test feature vector $\mathbf{x}_t$ from the corresponding probability values:

$$\hat{\mathbf{y}}_t = f(\mathbf{x}_t) = (Pr[\mathbf{y}_t = (1, 0)], Pr[\mathbf{y}_t = (0, 1)]) \quad (8)$$

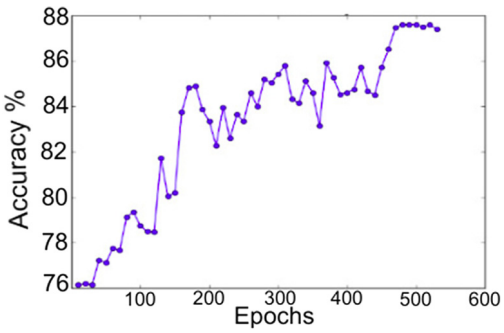

Fig. 9. Avg. accuracy for 10-fold cross-validation (as the algorithm converges vs epochs).

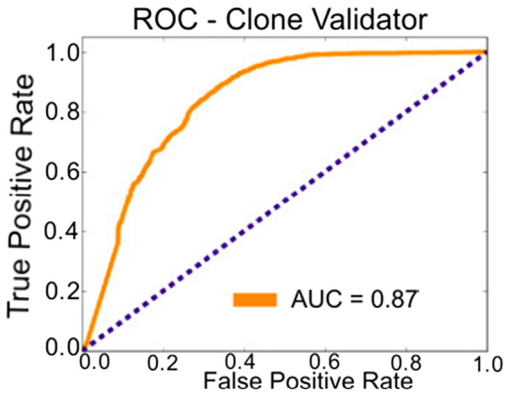

Fig. 10. ROC — Curve for validation by the method.

where, $Pr[\mathbf{y}_t = (1, 0)]$ denotes the probability of the test code clone with feature $\mathbf{x}_t$ to be true positive. 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 (as discussed in Section 3.4).

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. The model was trained and tested using 10-fold cross validation. Fig. 9, 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 for $k = 107$, giving an accuracy of 87.4%.

To analyze the output quality across different values of γ we plotted the ROC curve, which is shown in Fig. 10. The calculated Area Under the Curve (AUC) for the ROC curve is found to be 0.87. From the ROC curve, the proposed method can recommend the Θ 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 results while training. Besides Fig. 11, shows the Precision–Recall curve for the proposed method for varying $\gamma[0,1]$ values. In case of Precision–Recall curve, the AUC found to be 0.85.

5.4. Deep learning

In addition to the Backpropagation Neural Network, we also explored extending the model to deep learning for training the prediction function $f(\mathbf{x})$, for mapping $\mathbb{R}^n$ to $\mathbb{R}^l$, where n and l are dimensions of feature vector and class labels respectively. As per Eq. (7), we applied the Rectified Linear Unit (ReLU), $\sigma(z) = \max(0, z)$ and Sigmoid function, $\sigma(z) = \frac{1}{1+\exp(-z)}$ as the activation functions for the hidden and output layers respectively.

Table 2
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
Naive 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
Naive 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

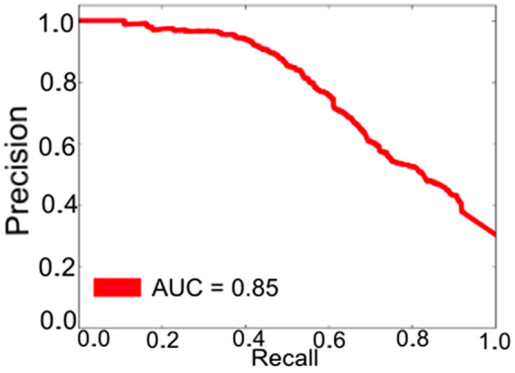

Fig. 11. PR — Curve for validation by the method.

Deep Neural Networks with a large number of layers and nodes often face an overfitting problem (Srivastava et al., 2014). In these cases, some specific set of neurons adapts too much in the decision while ignoring a large set of other neurons, and thus failing to generalize the learned classification model. Srivastava et al. (2014) proposed a dropout method for preventing neural networks from such an overfitting problem. This method drops out random neurons along with their corresponding connections forming thinned networks in the training phase preventing too much co-adaption. The network is then approximated from the thinned networks in the testing phase.

We used Keras (2015) - a python deep learning library, running on top of TensorFlow (Abadi et al., 2016) - a recent open source project released by Google for deep learning. We investigated the model's clone classification performance with different configurations (e.g. different dropout probability, learning nodes, hidden layers). From our investigation, we got a comparatively better result with the sequential stacking of three layers in addition to the input and output layers. We used dropout probability $p = 0.5$ (e.g. a neuron along with its corresponding connection is dropped out with a probability of 0.5), giving generality in training the model. With 32 neurons in each of the hidden layers, the obtained accuracy was 76%.

The data distribution study in Section 4.3, indicated the non-linear function requirement for clone classification. Hence from the findings, we investigated different machine learning models for the classification. The study helps better understanding the classification problem and also can contribute to future research works in this area for building even better models from our insights. We got the best classification result using Backpropagation Neural Networks. To verify the classification performance for different use cases, we have performed several experiments with this machine learning model. The detail study findings have been presented in Section 6.

6. Experiments

6.1. Implementation details

We implemented a prototype[3] of the system for testing the performance of the proposed method in different experimental setups. For collecting the user-specific training data, a cloud-based web application was first developed as shown in Fig. 12. For getting the automatic validation results, we also developed a cloud based web application (i.e., in addition to the command line based support) for easier user accessibility (as shown in Fig. 13). We used Python 2.7, as the server-side language. The web application was developed using Flask (2015) - 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. For a given code clone pair, user decisions (true positive or false positive) are then stored in the server database, mapping against the corresponding user profile. We used CouchDB (CouchDB, 2010) - a NoSQL database system, that supports easier scaling up and distributed computing for Big Data (Moniruzzaman and Hossain, 2013). We selected CouchDB to take advantage of this feature of the database for handling a large amount of code clones in our future works.

The collected manual validated code clone pairs are then used for building the classification model as described in the proposed method. We used TXL (Cordy et al., 1991) for pre-processing the code clone pairs for extracting several features (as described in Section 4.3) for training the model. Following the pre-processing, the clone features were then extracted using the Java programming language. The feature vectors from the training dataset were then used for learning the classification model. We experimented with several machine learning algorithms for the clone classification. The details of the findings and comparison study among the classifiers have been presented in Section 5. The trained model is then used for user-specific validation of new code clone pairs. The reported code clone pairs for validation can be sent to the cloud, where the trained model predicts and returns the validation score to the corresponding user end. We also implemented a prototype for receiving the test code clone pairs and sending the validation score from trained model to the user end. As shown in Fig. 14, the server is requested with code clone pairs for validation, server then uses the trained machine learning model for clone classification and then sends back the validation score to the user in JSON format. The server requests and responses are done using the RESTful API. The validation score then can be used in the corresponding code clone detection tools for classification or comprehension of the clones as per the user configuration. In addition to this cloud deployment, the modular trained model can

[3] https://github.com/pseudoPixels/CloneCognition.

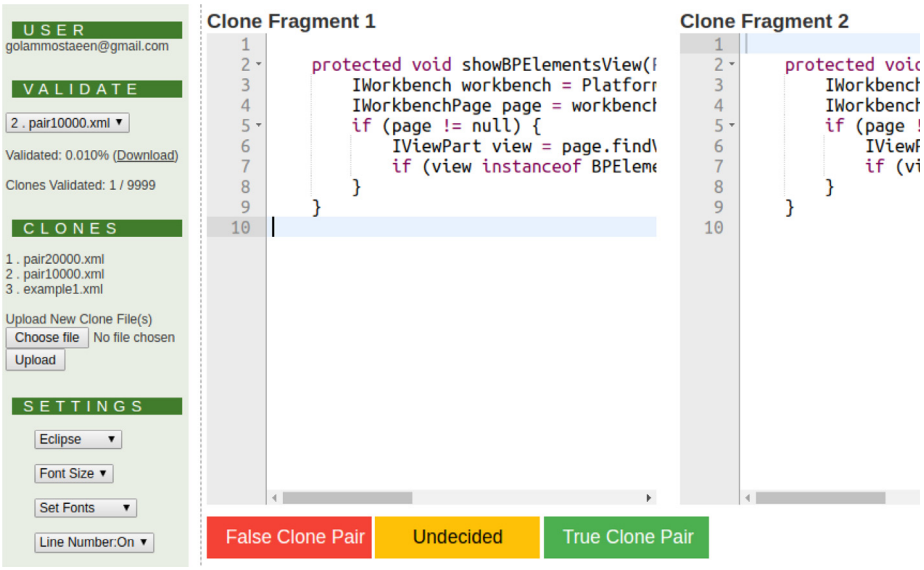

Fig. 12. System prototype: Systems clone validation.

Table 3
Information of the open source projects in the experimentation.

Software system	LoC[a]	Clone pairs	Average lines[b]	Average tokens[b]
Luaj	**36155**	1181	15	79
Ucdetector	4388	183	11	67
Autocover tool	3989	150	13	50
Upm-swing	**13243**	561	11	73
ipscan	7082	323	10	58
JavaGB	24211	1246	9	58
JavaOcr	7699	208	18	**90**
JavaFileManager	**25898**	1017	12	68
jMemorize	13109	598	10	44
FileBot	18369	834	11	59
JAIMBot	14096	583	12	**83**
JLipSync	3671	64	28	158

[a]Some of results are combination of detected clones from multiple clone detection tools (as listed in Table 4).
[b]Average per code clone fragment.

also be used locally for prediction by embedding with particular clone detection tools.[4]

6.2. Experimental setup

Automatic clone validation can contribute to clone analysis across different scenarios and requirements starting from smaller to large scale software system. For this reason, we were interested in evaluating the system across different environmental set ups: with several clone detection tools, users, and software systems. Table 3 lists a set of open-source projects that we used in evaluating the proposed system. For testing the model generality, we also used multiple clone detection tools on those software systems for detecting code clones. Table 4 shows the clone detection tools along with their used configuration for the study.

6.3. 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 varies significantly from person to person (Keivanloo et al., 2015; Charpentier et al., 2015). Thus, in order to get more concrete information about the validation accuracy of the trained model, we were interested in 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 the Mutation Framework (Svajlenko et al., 2013) for creating such a code clone benchmark. The framework takes a code fragment as input and 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 5. These operations create three different types (Type-1, Type-2 and Type-3) of true positive clones which are mostly simpler, straight forward and have minimal subjective bias. We used different original code fragments from BigCloneBench (Svajlenko et al., 2014) 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 the dataset as described in Section 4.1. 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 6. The possible reasoning for this is that though the artificially created clones contain minimal subjective biases, they have one disadvantage: they are very similar with one another and comparatively easily distinguishable (as also noticeable from higher recall value in Table 6).

6.4. 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 5. 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. The used open source software projects have been listed with details in Table 3. We used different code clone detection tools for detecting the code clones available in those open source software

[4] The prototype system implementation and source codes will be made available for usage and research purpose after the blind review process.

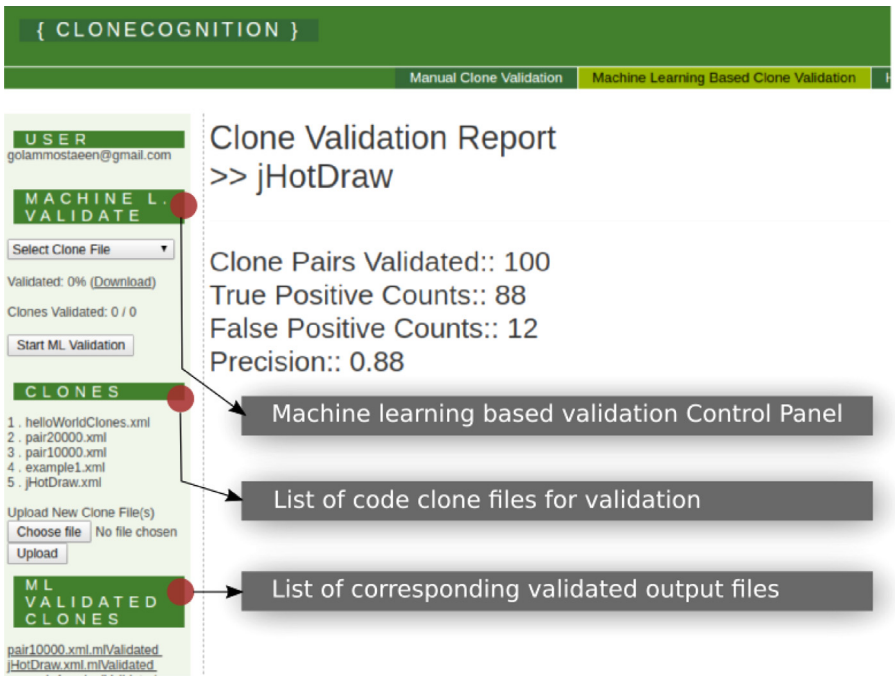

Fig. 13. System prototype: Validated clone report.

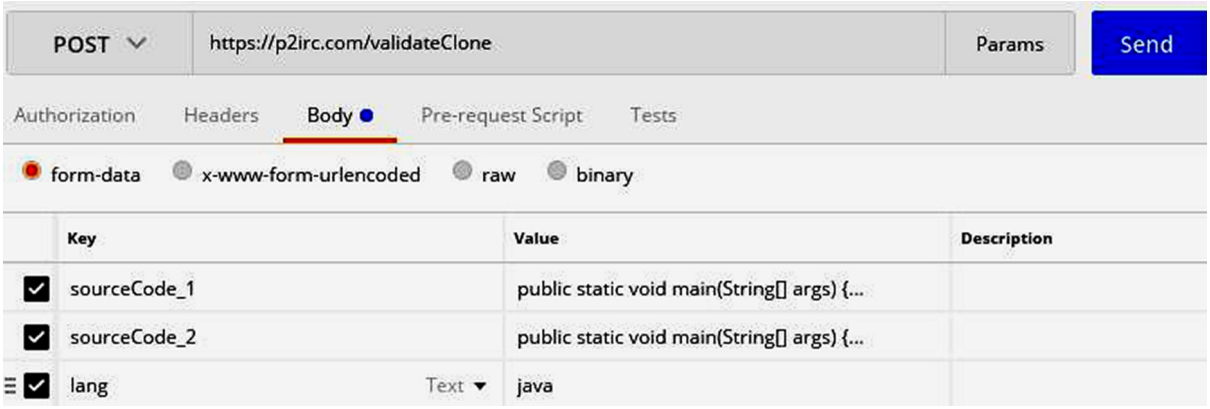

Fig. 14. System prototype: Getting prediction response from cloud.

project. We used 6 different code clone detection tools to test the generality of the proposed method. Table 4 lists the code clone detection tools used along with their corresponding version number and used configurations. The reported code clones from code clone detection tools were then manually validated by different users. None of the users were previously involved in building the training dataset as discussed in Section 4.1. Besides, to compare the performance of the proposed method with similar existing method — FICA (Yang et al., 2015), we contacted and got the source code[5] from the corresponding authors.

The trained model was used for predicting the user clone validation for each of the projects. Fig. 15 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 noticeably lower performance. 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-wide precision and recall were calculated for the approaches. The obtained result has been presented in Table 7. 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 noticeable in the box plot in Fig. 16. 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 a relatively better and more 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 36,155 with an average of 79 tokens per clone fragment, resulting in the classification to take noticeably longer time than the proposed approach. We got

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

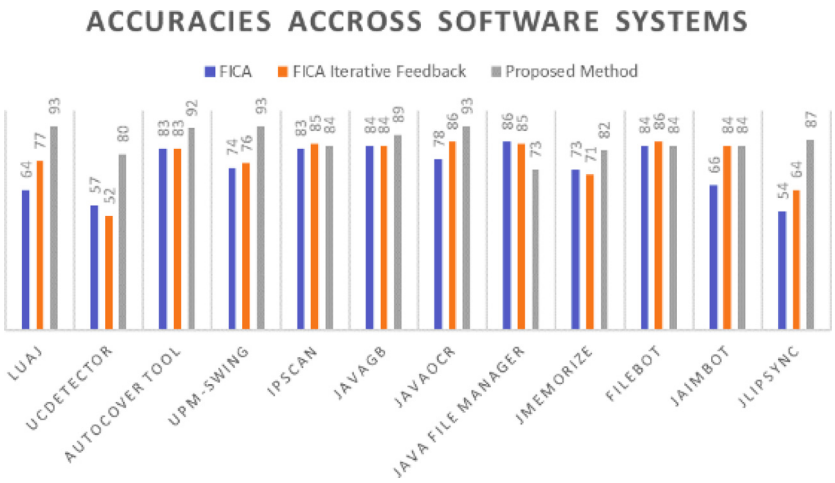

Fig. 15. Accuracy comparison of the methods across different software systems.

Table 4
Used clone detection tools for the study.

CDT	Ver.	Tool configuration
iClones (Göde and Koschke, 2009)	0.2	mintokens = 50, minblock = 20
NiCad (Roy and Cordy, 2008)	4.0	blocks, 30%, 6-2500 lines, blind-renaming, abstract-literal
SimCad (Uddin et al., 2013)	2.2	generous, 6+ lines, blocks
CloneWorks (Svajlenko and Roy, 2017)	0.2	Type-3 Aggressive, 6 lines, blocks
Simian (Simian, 2010)	2.4	6 lines, ignore overlapping blocks, balances parentheses
Ctcompare (Toomey, 2012)	3.2	50 tokens, 3 replacements

Table 5
Operations used to create artificial code clones via mutation framework (Svajlenko et al., 2013).

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 6
Result on artificial code clones.

Accuracy	Precision	Recall	F1-score
90%	0.89	0.99	0.93

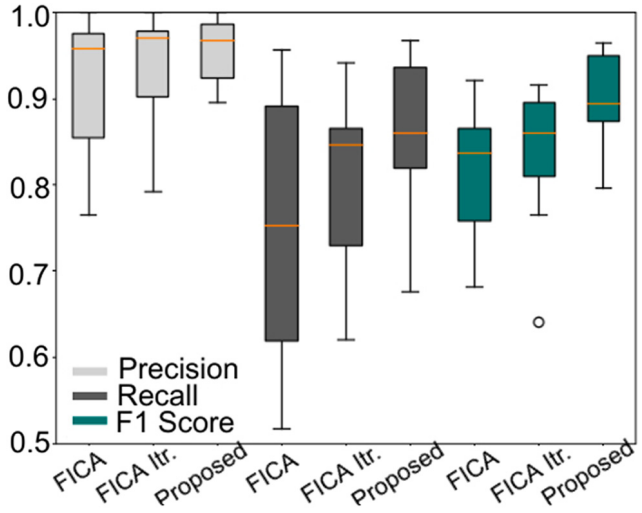

Fig. 16. Result quality comparison of the methods.

the same behavior for similar relatively bigger software system like: 'Java File Manager' and 'Upm-swing'.

7. Result discussion

Artificial Neural Networks are efficient computing models which can approximate complex functions. Different variants of Artificial Neural Networks have been widely used for pattern recognition in different branches of computer science (Cho, 1997; Looney, 1997; Rocha and Silva, 2016). On the other hand, one of the major criticisms is their being black boxes, since no satisfactory explanation of their behavior 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, 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 a 'black box' in the middle of input sets and its predicted decision we tried to find out if there is any bias 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. Fig. 17, shows the 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

Table 7
Comparison with existing systems.

Software system	FICA		FICA iterative		Proposed method	
	Precision	Recall	Precision	Recall	Precision	Recall
Luaj	0.969642857	0.629930394	0.97619047	0.769230769	0.979827089	0.945319741
Ucdetector	0.951219512	**0.549295775**	0.971428571	**0.478873239**	0.895833333	0.883561644
Autocover tool	0.830188679	0.956521739	0.843137255	0.934782609	0.926315789	0.967032967
Upm-swing	0.989690722	0.738461538	0.994923858	0.753846154	0.985971944	0.944337812
ipscan	0.863247863	0.918181818	0.922330097	0.863636364	0.964912281	0.800970874
JavaGB	0.784722222	0.875968992	0.792114695	0.856589147	**0.9**	0.861878453
JavaOcr	0.970588235	0.76744186	0.973684211	0.860465116	0.988304094	0.933701657
JavaFileManager	0.962962963	0.882352941	0.967254408	0.868778281	0.941807044	0.725235849
jMemorize	0.926829268	0.619565217	0.933774834	0.658878505	0.91576087	0.828009828
FileBot	0.765217391	0.946236559	0.791855204	0.940860215	**0.969581749**	0.676392573
JAIMBot	0.993710692	**0.619607843**	0.98156682	0.835294118	0.987980769	0.825301205
JLipSync	1	0.517241379	1	0.620689655	1	0.857142857

* Some of results are combination of detected clones from multiple clone detection tools (as listed in Table 4).
* Average per code clone fragment.

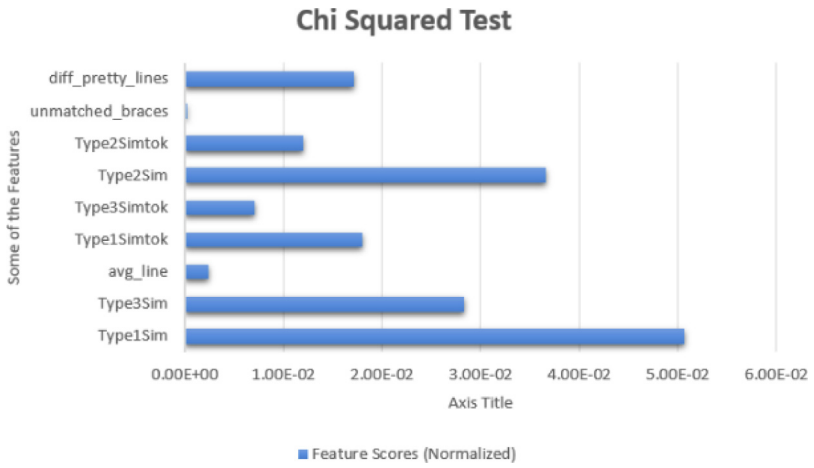

Fig. 17. Feature score via Chi-Squared test.

features rather than being completely dominated by one or more features. This gives us some idea that the trained model is not noticeably biased towards 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 logical for the stated clone validation problem. Similarly, average code clone fragment size or unmatched braces has much less contribution score as discussed in Section 4.3. The 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 a 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 types of clones or there can also be possibility that the proposed method fails to validate all the Type 1 clones. 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 is 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 Fig. 18. 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 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 aligned along a particular axis, such as 'Type 3 Clone Similarity'. 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 clearly noticeable. From those studies on the misclassified test samples by the proposed method, we can conclude that it did not fail for any particular type of clone. Similarly, the proposed method can successfully classify all three types of code clones as we can notice from the randomness of the correctly classified test samples in the 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. The Neural Network converged within a range of 500 to 600 epochs. Validation accuracy given by the proposed method is found to be 87.4% after averaging each of the 10 folds testing. Besides the trained system was tested by completely different software systems. We found the proposed

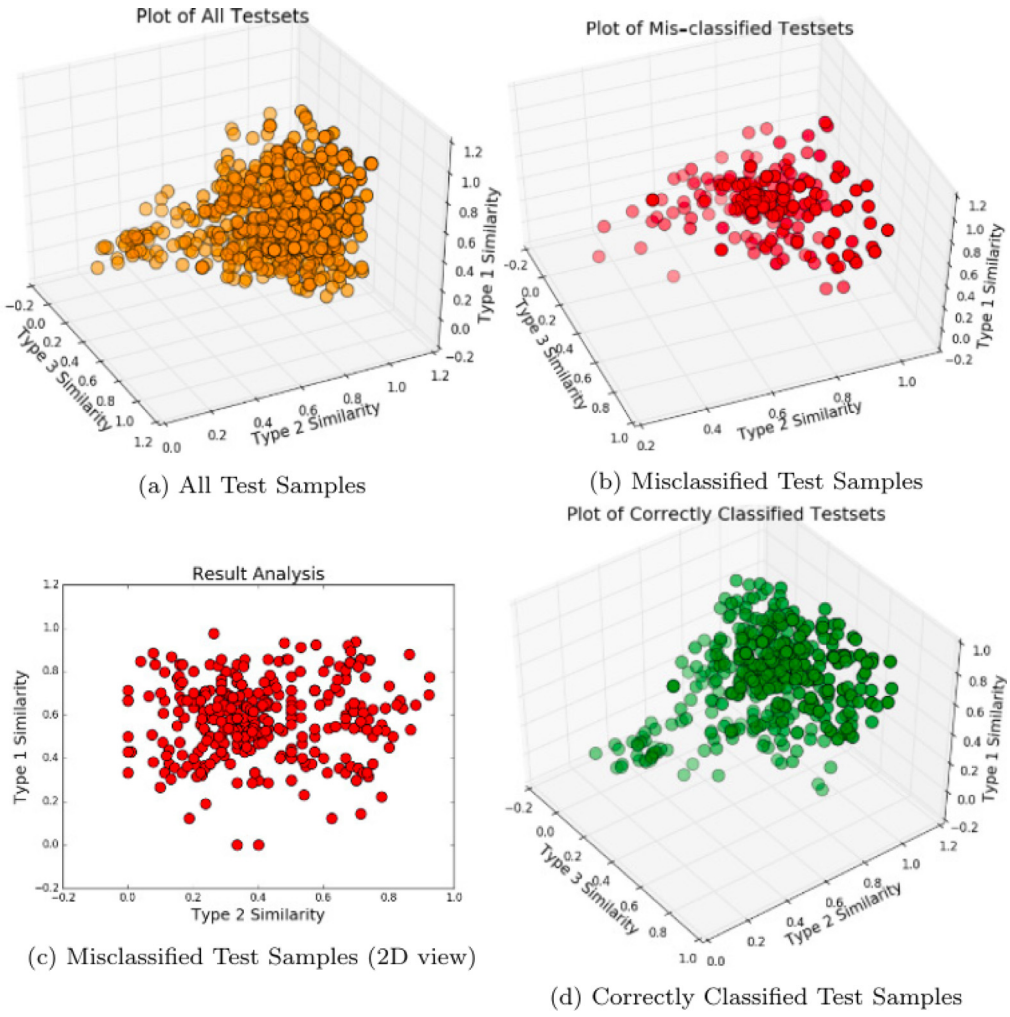

Fig. 18. Classification result analysis for different types of code clones.

method to come up with promising accuracies for clone validation. These positive results reveal opportunities for using machine learning for clone validation to assist in overall code clone maintenance and analysis process.

Answering RQ 2, (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 that the method does not fail for any particular type of clones. From the plotting of correctly classified or validated clones by the proposed method in Fig. 18, 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 clone. These results demonstrate 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 Fig. 15, the validation result in conjunction with different tools found to be promising.

8. Threats to validity

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

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

threat to validity by taking the validation decision from multiple programmers.

Another likely threat to validity is the 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 work 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 of having any such errors in feature calculations to make the evaluation as accurate as possible.

9. Conclusion

In this paper, we introduced a machine learning based 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 large number of clones and often does not consider the preferences of user's opinion or requirement. This leads to manual validation of the result from the clone detection tools which gets worse for large-scale software systems. We have proposed a machine learning approach that assists in automatic validation of 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 projects. We found promising accuracy with the automatic validation of clones by the proposed method.

Acknowledgements

This research is supported by the Natural Sciences and Engineering Research Council of Canada (NSERC), and by two Canada First Research Excellence Fund (CFREF) grants coordinated by the Global Institute for Food Security (GIFS) and the Global Institute for Water Security (GIWS).

References

Abadi, Martín, Agarwal, Ashish, Barham, Paul, Brevdo, Eugene, Chen, Zhifeng, Citro, Craig, Corrado, Greg S, Davis, Andy, Dean, Jeffrey, Devin, Matthieu, et al., 2016. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. ArXiv preprint arXiv:1603.04467.

Ambient Software Evoluton Group, 2013. IJaDataset 2.0. http://secold.org/projects/seclone.

Baker, Brenda S., 1993. A program for identifying duplicated code. In: Computing Science and Statistics. p. 49, Proceedings Published by Various Publishers.

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

Bellon, Stefan, Koschke, Rainer, Antoniol, Giulio, Krinke, Jens, Merlo, Ettore, 2007. Comparison and evaluation of clone detection tools. IEEE Trans. Softw. Eng. 33 (9).

Bettenburg, Nicolas, Shang, Weyi, Ibrahim, Walid, Adams, Bram, Zou, Ying, Hassan, Ahmed E, 2009. An empirical study on inconsistent changes to code clones at release level. In: Reverse Engineering, 2009. WCRE'09. 16th Working Conference on. IEEE, pp. 85–94.

Breiman, Leo, 2001. Random forests. Mach. Learn. 45 (1), 5–32.

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

Cho, Sung-Bae, 1997. Neural-network classifiers for recognizing totally unconstrained handwritten numerals. IEEE Trans. Neural Netw. 8 (1), 43–53.

Cordy, James R., Halpern-Hamu, Charles D., Promislow, Eric, 1991. TXL: A rapid prototyping system for programming language dialects. Comput. Lang. 16 (1), 97–107.

CouchDB, 2010. Apache Software Foundation. CouchDB - NoSQL Database System. http://couchdb.apache.org/.

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

Duala-Ekoko, Ekwa, Robillard, Martin P., 2007. Tracking code clones in evolving software. In: Proceedings of the 29th International Conference on Software Engineering. IEEE Computer Society, pp. 158–167.

Flask, 2015. A Microframework for Python based on Werkzeug, Jinja 2. http://flask.pocoo.org/.

Friedman, Nir, Geiger, Dan, Goldszmidt, Moises, 1997. Bayesian Network classifiers. Mach. Learn. 29 (2–3), 131–163.

Göde, Nils, Koschke, Rainer, 2009. Incremental clone detection. In: Software Maintenance and Reengineering, 2009. CSMR'09. 13th European Conference on. IEEE, pp. 219–228.

Göde, Nils, Koschke, Rainer, 2011. Frequency and risks of changes to clones. In: Proceedings of the 33rd International Conference on Software Engineering. ACM, pp. 311–320.

Hall, Mark, Frank, Eibe, Holmes, Geoffrey, Pfahringer, Bernhard, Reutemann, Peter, Witten, Ian H, 2009. The weka data mining software: an update. ACM SIGKDD explorations newsletter 11 (1), 10–18.

Higo, Yoshiki, Ueda, Yasushi, Kamiya, Toshihro, Kusumoto, Shinji, Inoue, Katsuro, 2002. On software maintenance process improvement based on code clone analysis. In: Product Focused Software Process Improvement. Springer, pp. 185–197.

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

Jiang, Lingxiao, Misherghi, Ghassan, Su, Zhendong, Glondu, Stephane, 2007. Deckard: Scalable and accurate tree-based detection of code clones. In: Proceedings of the 29th International Conference on Software Engineering. IEEE Computer Society, pp. 96–105.

John, George H., Langley, Pat, 1995. Estimating continuous distributions in Bayesian classifiers. In: Proceedings of the Eleventh Conference on Uncertainty in Artificial Intelligence. Morgan Kaufmann Publishers Inc., pp. 338–345.

Juergens, Elmar, Deissenboeck, Florian, Hummel, Benjamin, Wagner, Stefan, 2009. Do code clones matter? In: Software Engineering, 2009. ICSE 2009. IEEE 31st International Conference on. IEEE, pp. 485–495.

Kamiya, Toshihiro, Kusumoto, Shinji, Inoue, Katsuro, 2002. Ccfinder: a multilinguistic token-based code clone detection system for large scale source code. IEEE Trans. Softw. Eng. 28 (7), 654–670.

Kapser, Cory, Godfrey, Michael W., 2004. Aiding comprehension of cloning through categorization. In: Software Evolution, 2004. Proceedings. 7th International Workshop on Principles of. IEEE, pp. 85–94.

Kapser, Cory, Godfrey, Michael W., 2006a. "Cloning considered harmful" considered harmful. In: Reverse Engineering, 2006. WCRE'06. 13th Working Conference on. IEEE, pp. 19–28.

Kapser, Cory J., Godfrey, Michael W., 2006b. Supporting the analysis of clones in software systems. J. Softw.: Evol. Process 18 (2), 61–82.

Keivanloo, Iman, Zhang, Feng, Zou, Ying, 2015. 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. IEEE, pp. 201–210.

Keras, 2015. Keras: The Python Deep Learning library. https://keras.io/.

Koschke, Rainer, Falke, Raimar, Frenzel, Pierre, 2006. Clone detection using abstract syntax suffix trees. In: Reverse Engineering, 2006. WCRE'06. 13th Working Conference on. IEEE, pp. 253–262.

Lam, Wai, Bacchus, Fahiem, 1994. Learning Bayesian belief networks: An approach based on the mdl principle. Comput. Intell. 10 (3), 269–293.

Li, Zhenmin, Lu, Shan, Myagmar, Suvda, Zhou, Yuanyuan, 2006. CP-Miner: Finding copy-paste and related bugs in large-scale software code. IEEE Trans. Softw. Eng. 32 (3), 176–192.

Looney, Carl Grant, 1997. Pattern Recognition Using Neural Networks: Theory and Algorithms for Engineers and Scientists. Oxford University Press, Inc.

Mondal, Manishankar, Rahman, Md Saidur, Roy, Chanchal K, Schneider, Kevin A, 2017a. Is cloned code really stable? Empir. Softw. Eng. 1–78.

Mondal, M., Rahman, M. S., Saha, R. K., Roy, C. K., Krinke, J., Schneider, K. A., 2011. An empirical study of the impacts of clones in software maintenance. In: 2011 IEEE 19th International Conference on Program Comprehension, pp. 242–245.

Mondal, M., Roy, C. K., Schneider, K. A., 2014. Automatic ranking of clones for refactoring through mining association rules. In: 2014 Software Evolution Week - IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering. CSMR-WCRE, pp. 114–123.

Mondal, M., Roy, C. K., Schneider, K. A., 2014. Automatic identification of important clones for refactoring and tracking. In: 2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation, pp. 11–20.

Mondal, Manishankar, Roy, Chanchal K., Schneider, Kevin A., 2017b. Bug propagation through code cloning: An empirical study. In: Software Maintenance and Evolution (ICSME), 2017 IEEE International Conference on. IEEE, pp. 227–237.

Moniruzzaman, A.B.M., Hossain, Syed Akhter, 2013. Nosql database: New era of databases for big data analytics-classification, characteristics and comparison. ArXiv preprint arXiv:1307.0191.

Pan, Sinno Jialin, Yang, Qiang, 2009. A survey on transfer learning. IEEE Trans. knowl. Data Eng. 22 (10), 1345–1359.
Quinlan, Ross, 1993. C4.5: Programs for Machine Learning. Morgan Kaufmann Publishers, San Mateo, CA.
Rocha, Priscila Lima, Silva, Washington Luis Santos, 2016. 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. IEEE, pp. 46–51.
Roy, Chanchal Kumar, Cordy, James R., 2007. A survey on software clone detection research. Queen's School Comput. TR 541 (115), 64–68.
Roy, Chanchal K., Cordy, James R., 2008. 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. IEEE, pp. 172–181.
Saha, R., Roy, C., Schneider, K., Perry, D., 2013. Understanding the evolution of type-3 clones: an exploratory study. In: MSR. pp. 139–148.
Saini, Vaibhav, Farmahinifarahani, Farima, Lu, Yadong, Yang, Di, Martins, Pedro, Sajnani, Hitesh, Baldi, Pierre, Lopes, Cristina V, 2019. Towards automating precision studies of clone detectors. In: 2019 IEEE/ACM 41st International Conference on Software Engineering. ICSE. IEEE, pp. 49–59.
Sajnani, Hitesh, Saini, Vaibhav, Svajlenko, Jeffrey, Roy, Chanchal K, Lopes, Cristina V, 2016. Sourcerercc: Scaling code clone detection to big-code. In: Software Engineering (ICSE), 2016 IEEE/ACM 38th International Conference on. IEEE, pp. 1157–1168.
Simian, 2010. Code Clone Detection Tool. http://www.redhillconsulting.com.au/products/simian/.
Srivastava, Nitish, Hinton, Geoffrey E, Krizhevsky, Alex, Sutskever, Ilya, Salakhutdinov, Ruslan, 2014. Dropout: a simple way to prevent neural networks from overfitting. J. Mach. Learn. Res. 15 (1), 1929–1958.
Svajlenko, Jeffrey, Islam, Judith F, Keivanloo, Iman, Roy, Chanchal K, Mia, Mohammad Mamun, 2014. Towards a big data curated benchmark of inter-project code clones. In: Software Maintenance and Evolution (ICSME), 2014 IEEE International Conference on. IEEE, pp. 476–480.
Svajlenko, Jeffrey, Roy, Chanchal K., 2016. A machine learning based approach for evaluating clone detection tools for a generalized and accurate precision. Int. J. Softw. Eng. Knowl. Eng. 26 (09n10), 1399–1429.
Svajlenko, Jeffrey, Roy, Chanchal K., 2017. Cloneworks: a fast and flexible large-scale near-miss clone detection tool. In: Proceedings of the 39th International Conference on Software Engineering Companion. IEEE Press, pp. 177–179.
Svajlenko, Jeffrey, Roy, Chanchal K., Cordy, James R., 2013. A mutation analysis based benchmarking framework for clone detectors. In: Software Clones (IWSC), 2013 7th International Workshop on. IEEE, pp. 8–9.
Tairas, Robert, Gray, Jeff, 2006. Phoenix-based clone detection using suffix trees. In: Proceedings of the 44th Annual Southeast Regional Conference. ACM, pp. 679–684.
Tairas, Robert, Gray, Jeff, 2009. An information retrieval process to aid in the analysis of code clones. Empir. Softw. Eng. 14 (1), 33–56.
Toomey, Warren, 2012. Ctcompare: Code clone detection using hashed token sequences. In: Software Clones (IWSC), 2012 6th International Workshop on. IEEE, pp. 92–93.
Torrey, Lisa, Shavlik, Jude, 2010. Transfer learning. In: Handbook of Research on Machine Learning Applications and Trends: Algorithms, Methods, and Techniques. IGI Global, pp. 242–264.
Uddin, Md Sharif, Roy, Chanchal K., Schneider, Kevin A., 2013. Simcad: An extensible and faster clone detection tool for large scale software systems. In: Program Comprehension (ICPC), 2013 IEEE 21st International Conference on. IEEE, pp. 236–238.
Yang, Jiachen, Hotta, Keisuke, Higo, Yoshiki, Igaki, Hiroshi, Kusumoto, Shinji, 2015. Classification model for code clones based on machine learning. Empir. Softw. Eng. 20 (4), 1095–1125.
Zibran, Minhaz F., Roy, Chanchal K., 2011. Towards flexible code clone detection, management, and refactoring in IDE. In: Proceedings of the 5th International Workshop on Software Clones. IWSC, pp. 75–76.
Zibran, Minhaz F., Roy, Chanchal K., 2012a. IDE-based real-time focused search for near-miss clones. In: Proceedings of the 27th Annual ACM Symposium on Applied Computing. SAC, pp. 1235–1242.
Zibran, M., Roy, C., 2012b. IDE-based real-time focused search for near-miss clones. In: ACM-SAC (SE Track). pp. 1235–1242.