Improving Maintenance-Consistency Prediction During Code Clone Creation

FANLONG ZHANG\(^1\), SIAU-CHENG KHOO\(^2\), AND XIAOHONG SU\(^3\)

\(^1\)School of Computer, Guangdong University of Technology, Guangzhou 510006, China
\(^2\)School of Computing, National University of Singapore, Singapore 119077
\(^3\)School of Computer Science and Technology, Harbin Institute of Technology, Harbin 150001, China

Corresponding author: Fanlong Zhang (zhangfanlong@gdut.edu.cn)

This work was supported in part by the Guangdong Provincial Key Laboratory of Cyber-Physical Systems under Grant 2016B030301008, in part by the National and Local Joint Engineering Research Center of Intelligent Manufacturing Cyber-Physical Systems, in part by the Key Area Research and Development Program of Guangdong under Grant 2018B010109007, in part by the National Natural Science Foundation of China under Grant 61672191, and in part by the 13th Five-Year National Science and Technology Major Project of China under Grant 2017YFC0702204.

ABSTRACT Developers frequently introduce code clones into software through the copy-and-paste operations during the software development phase in order to shorten development time. Not all such clone creations are beneficial to software maintenance, as they may introduce extra effort at the software maintenance phase, where additional care is needed to ensure consistent change among these clones; i.e., changes made to a piece of code may need to be propagated to other clones. Failure in doing so may risk introducing bugs into software, which are usually called consistent defect. In response to the rampant maintenance cost caused by the introduction of new clones, some researchers have advocated the use of machine-learning approach to predict the likelihood of consistent change requirement when clones are freshly introduced. Leading in this approach is the work by Wang et al., which uses Bayesian Network to model maintenance-consistency of newly introduced clones. In this work, we leverage the success of the above-mentioned work by providing a revised set of attributes that has been shown to strengthen the predictive power of the Bayesian network model, as determined more quantitatively by the precision and recall levels. We firstly provide the definition of clone consistency-maintenance requirement, which can help transfer this problem to a classification problem. Then, based on collecting all clone creation operations through traversing clone genealogies, we redesign the attribute sets for representing clone creation with more information in code and context perspective. We evaluate the effectiveness of our approach on four open source projects with more quantitative analysis, and the experimental results show that our approach possesses a powerful ability in predicting clone consistency. To transfer this work into practice, we develop an Eclipse plug-in tool of this prediction to aid developers in software development and maintenance.

INDEX TERMS Code clones, clone consistent change, clone consistency prediction, software maintenance, machine learning.

I. INTRODUCTION

The ever-increasing demand for shortening the development time to reach the marketplace has put immense pressure on software developers to find means to develop their codes quickly. One technique that is used prevalently, and even encouraged in school, is to reuse existing code fragments. Among these reuses, copy-and-paste operation is by far the simplest and thus widely used technique. After a piece of code has been copied and pasted, the developer may decide to modify it slightly – or not at all, thus creating many similar code fragments within a piece of software. Two pieces of similar codes are known as clones to each other [1], and operations such as “copy-and-paste” are called clone creating operation. Depending on the “similarity” criteria, two code clones may be identical, syntactically or semantically similar.

As software evolves, these code clones may change over time; they may become totally dissimilar, or they may continue to remain similar, and sometime being modified in tandem. This process of “clone evolution” has been modeled...
nicely by Kim et al. with the concept of clone genealogy [2]. In this genealogy, clones that are similar to one another are grouped together to form clone groups. It has been shown that frequent changes on code clones do occur during software evolution [3]. Within a clone group, changes made to one piece of code clone may require to be propagated to other clones within the group, thus remaining to be similar. Such changes are usually termed as consistent change [4]. Consistent changes of code clones bring forth additional cost in software maintenance, as developers will need to ensure clones within a group are changed consistently [5]; failing which may likely introduce bugs into the software [6], which are termed clone consistent defect. Currently, in order to avoid such defects, developers need to examine the corresponding code clones individually and determine if consistent changes need to be exerted on them every time when one of them is changed. Ideally, some tools that can assist developers in determining the need for consistent changes will help save software maintenance cost [7], [8].

In this work, we aim at helping developers to predict the likelihood that a clone creating operation will lead to a consistent change in the associated clone group in future clone evolution. We say that a clone creating operation meets the clone consistency-requirement if it can result in consistent change in future evolution of clones. Specifically, when developers create code clones – such as through the copy-and-paste operation, we alert them of the likelihood that the newly formed clone group may likely require consistent change in future. With this knowledge, developers can re-think if they would like to perform the clone creation, thus improving development efficiency. Pioneering in this exploration of predicting clone-consistency-requirement is the work done by Wang et al. [7]. In their work, Wang et al. predict clone consistency by considering three attribute sets, drawing from the following three perspectives: history, code, and context. However, we identify three shortcomings in their work. Firstly, the historical attributes have a weak correlation with clone creating operation. It can become a hurdle when performing prediction because it requires the availability of (some summary of) all source code of software’s repository. Secondly, the rest of the attributes do not adequately capture information about the codes and their contexts. Lastly, they do not provide any means to demonstrate how the technique can be applied in practice. Therefore, practical application of this method can be limited and cannot be directly applied to specific software development practice.

In this work, we propose a revised set of attributes that can improve the predictability of maintenance consistency-requirement during software development phase. We first revise the definition for clone consistency-requirement. We then build clone genealogies for software’s repository to collect all clone-creation operations. Then, we revise and extend code and context attribute sets to represent the copied and pasted code clones respectively in this clone-creation environment as well as removal of the historical attributes.\(^1\) We next employ Bayesian network [9] as our prediction model for predicting clone maintenance consistency, and offer an eclipse plug-in that helps developers predict clone consistency in practice. We conduct empirical evaluation of our system on four projects; the results demonstrates that our prediction model can be effective in prediction, with high precision and recall rates. The contributions of this paper are as follows:

- We propose an approach to predict clone maintenance consistency from the occurrence of code clones’ creating operation, as well as definitions for the code clone consistent change and its consistency-requirement of clone creating operations;
- We extend and extract code and context attributes for maintenance consistency-requirement prediction. The results show that these attributes has positive impact on the ability of the predictor;
- We develop a prototype tool which is an Eclipse plug-in to aid developers in predicting clone maintenance consistency at clone creation time;
- We construct an evaluation of four projects, with results showing that our approach can predict clone consistency effectively with high precision and recall rate.

As our experiment on the open source projects indicates that our approach possesses effective predictive ability on clone maintenance consistency, using both code and context attribute sets, we encourage developers to employ all the code and context attribute sets described here when performing prediction in practice; In addition, the developers can also include other attributes to further sharpen the predictability of the model. On the other hand, because the effectiveness of the prediction model is highly dependent of the specific characteristics of the repository, we opine that developers should create different models to perform consistency prediction for different repositories.

The structure of this paper is as follows: Section 2 describes the related work in code clone research. Section 3 provides definitions in which code clone, the evolution model, as well as clone consistency-requirement. In Section 4, we describe in detail our maintenance consistency-requirement prediction modeled by Bayesian network. Section 5 details the implementation for our prototype tool as an eclipse plug-in. Section 6 describes our experimental projects and the experimental evaluation steps. We discuss and analyze the experimental results in Section 7. Section 8 describes any threats to the validity of this work and Section 9 concludes our work.

\(^1\) We believe that historical attributes are not needed for the prediction of consistency-requirement. Our reasoning is that there is no historical attribute information for clone creating instance. The historical attributes from Wang et al.’s work are not really for clone creating instance; rather, they record changes of the files that include code clones before the copy-and-paste operation. Moreover, their investigation into historical attributes indicates that it is feasible to use only code attributes and context attributes to predict clone consistency [7].
II. RELATED WORK

Research in code clone begins with the treatment of code clones as “bad smells”, where their presence is deemed to have adverse effect on the corresponding piece of software, and the corresponding recommended treatment of code clones is to eliminate them by refactoring technique [10]. This naturally leads to more in-depth discussion of whether code clones are considered harmful to the software. Proponents of “clones being harmful” believe that code clones can cause clone defects, leading to rise in software maintenance cost. For instance, Barbour et al. study the defect-prone of code clones, and find that code clones having inconsistent and divergent patterns are more likely to experience a defect [11]. Mondal et al. study the bug-proneness and late propagation tendency of code clones on different clone types, and find that Type-3 clones have the highest bug-proneness among three types of clones [12]. On the other hand, some researchers believe that code clones have beneficial effect to software quality. For instance, Krinke et al. investigate the life and changeability of code clones, and find that code clones have longer life span than non-clones, and are more stable than non-clones [13].

Taking into consideration the possible harmfulness of code clones, when code clones are about to be introduced into a piece of software, developers should try to avoid as much as possible those clones that may have adverse impact on software maintenance, and accept those which may not. This has triggered a flow of research into prediction of clone maintenance consistency – or “clone consistency” for short, which Wang et al. have pioneered; specifically, they predict the consistency maintenance requirement of code clones by extracting three different sets of attributes at copy-and-paste time [7]. Their work has an effective prediction results, though it still has some limitations when applied in practice, as described in the Introduction Section.

In our work, we leverage their work to better predict clone consistency by removing the historical attributes and enriching the other two attribute sets. In addition, Zhang et al. propose an approach to predict clone consistent change in a clone group when one piece of code clones in this group is modified. This is achieved by extracting the attribute sets from clone group perspective as well as introducing new attribute set from clone evolution perspective [8], [14].

To increase the awareness of the presence and evolution of code clones among developers, Kim et al. propose a novel concept of clone genealogy, that models the evolution of code clones alongside software revisions [2]. Conceptually, clone genealogy is a directed acyclic graph, in which each node in the graph represents a clone group; a path in the graph starting from a node (a clone group) depicts the evolution of code clones in this group throughout versioning of the software. Kim’s model of clone genealogy can fully describe the evolution of Type-1 and Type-2 clones; it is however not very suitable for capturing the evolution of Type-3 clones. Thus, Saha et al. propose a more elaborated model of code clones, especially for capturing consistent change of Type-3 clones [4]. Nevertheless, in the context of predicting clone maintenance-consistency, Kim’s model of consistent change can be imprecise, whereas Saha’s model can be too precise such that it cannot describe the maintenance cost of clone consistency as well. Therefore, in this work, we employ a more-balanced definition that is adapted to clone consistency prediction.

Research on clone detection can be dated back to 1990s, when a number of clone detection tools were developed, such as NiCad [15], CCFinder [16], iClone [17] and etc. Based on its usage, clone detection can be divided into text-based, token-based, tree-based, graph-based, etc. NiCad is a text-based clone detection tool, which can effectively detect Type-1, 2 and 3 of code clones [15]. In our work, we employ NiCad as the clone detection tool. Other possible detection tools are also available, such as ConQAT [18]. ConQAT is an open-source tool for executing software quality analysis, with clone detection functionality integrated. Next, CCFinder is a token-based detection tool; not only can it effectively validates clone codes, it can also measure code clone with some metrics from code perspective to help developers understand these detected code clones [16]. iClone is a tree-based clone detection tool that aims to find code clones in an incremental mode from software repository; it can detect code clones in the latter version of software repository based on the results of the last detection results of the version [17]. In order to help developers select suitable clone detection tools, researchers have reviewed 213 articles in the literature to assess the advantages and disadvantages of these techniques [19].

Recently, machine learning technique – especially deep learning technique – has been applied to clone detection. White et al. first propose an approach to detect code clones with deep learning; it employs RtNN and RvNN to extract the characteristic of source code from syntactic and semantic perspective [20]. Saini et al. develop a clone detection tool named Ore, which can not only detect Type1-3 code clones, but also detect Type-4 code clones in the Twilight Zone [21]. On the other hand, by focusing on function-level code clones, Li et al. propose the first solely token-based clone detection approach, which characterizes methods by vectors and feeds them to a deep neural network [22]. Moreover, Wei et al. propose an approach to detect function-level clones through extracting code characteristics with LSTM method [23]. Narayanan et al. propose an approach – named subgraph2vec – to learn code distribution representation from PDG, and detect code clones from software with such technique [24]. Sheneamer et al. propose a framework for clone detection, which can extract features to represent source code based on PDG and AST [25].

Clone detection and clone genealogy can only help developers beware of the presence of code clones in software. In order to solve the software maintenance problem brought forth by code clones, researchers have also turned their focus to clone maintenance and management research in recent years. The first clone maintenance technology was developed at the time when clones are viewed as “bad smell”, there,
researchers prefer to eliminate code clones from software by refactoring. For instance, Higo et al. propose an approach for refactoring code clones based on several metrics including the positional relationship in the class hierarchy, coupling between a code clone and its surrounding code [26]. Krishnan et al. propose approaches to find whether code clones can be refactored safely through parameterizing the differences between them [27], [28]. Later, researchers put forward more in-depth clone management research [29]. For instance, Nguyen et al. implement an eclipse plug-in JSync [30] that can be used to manage code clones in evolving software and can specially maintain automatically clone change consistently; this can be perceived as a possible follow-up step to be taken after consistent changes are discovered/predicted by our prediction model. Another tool, CeDAR, forwards clone detection results to a refactoring engine in Eclipse, that merges these different technologies into a clone management function [31], which can help developer remove code clones to avoid the maintenance cost caused by clone change. Note that none of these works focuses on predicting clone-consistent change; and the result of such prediction offered by our work can in fact compliment these works by providing consistency-change information to these works in a guided way.

III. DEFINITIONS

In this section, we first introduce related terminologies used in code clone research. Then, we give the definitions pertinent to clone maintenance consistency in terms of clone evolution, such as consistent change and its patterns.

A clone fragment is a piece of code fragment constituting a series of lines of code; Clone fragments can be collected by clone detection tools from software repository, and such detection tools will group code clones into clone groups. A clone group comprises several similar clone fragments, which are grouped together according to some similarity criterion. Based on the similarity between codes at syntax and semantics level, code clones can be classified into four categories, as follows: [32]:

- **Type-1 clone** is an exact clone; i.e., all clones in a clone group are identical without any modifications, except for differences in code layout and comments.
- **Type-2 clone** is a syntactically identical clone; i.e., all clones in a group are syntactically identical, differing only in the names of identifiers, types, literals, and methods.
- **Type-3 clone** is a syntactically similar clone; i.e., all clones in a group have some syntactic differences, such that one clone can be obtained from another clone in the same group by inserting, deleting, and replacing some statements.
- **Type-4 clone** is a semantically similar clone; i.e., all clones within a clone group have identical functionality.

To collect code clones, we employ NiCad [15] to detect code clones from the software. NiCad can detect Type-1, Type-2 and Type-3 code clones effectively. Through it, code fragments are reported as code clones when their similarity is greater than a threshold. Here, “similarity” is determined by the ratio of the number of identical lines of code between the two code fragments to the total code line. Given two code fragments $CF_1$ and $CF_2$, the similarity $Simtext(CF_1, CF_2) = 0.7$ indicates that the percentage of the number of identical lines of code between the two is 70%. In this paper, the similarity threshold of determining code clone is set to 70%, that is, two code fragments with similarity greater than 70% are deemed as code clones.

We employ the technique of clone genealogy proposed in [2] to model the evolution of code clones. Clone genealogy describes the evolution of a clone group since its inception and throughout the remaining life of the software. Formally,

**Clone Genealogy (CGE).** A directed acyclic graph originated from a clone group that describes its evolution throughout its entire life cycle within a software repository. A node in the graph represents a clone group, and an edge between two nodes denotes the evolution relationship of the clone group from one version to the next version in the software repository.

During clone evolution, clone fragments in a clone group may be modified by developer, possibly leading to a consistent change in the clone group. Consistent change will incur additional cost in software maintenance, as failure in maintaining such consistent change may lead to related defect. There are different clone consistent definitions according to different perspectives. The first one is from Kim’s definition [2], as follows:

**Consistent Change.** All clone fragments in the clone group have been changed consistently; thus all of them are again part of a clone group in next version.

This definition focuses on changes happening to Type-1 and Type-2 clones, such that the changed clones continue to be in the same clone group after some similar modification. Kim’s definition cannot be applied appropriately to Type-3 clones. Therefore, Saha et al. [4] propose a more rigorous definition that handles Type-3 clones, as follows:

**Consistent Change.** All clone fragments in a clone group have been changed consistently – within the minimum line/token numbers set by a subject clone detection tool; and thus all of them remain again as part of a clone group in next version.

This definition focuses on all clone fragments including Type-3 clones, and all clone fragments within a clone group are required to be modified in the same way, and still remain in this group after their modifications.

However, both the above two definitions are not well positioned for use in our clone consistency predictive work. Specifically, Kim’s definition fails to handle Type-3 clones, whereas Saha’s definition turns out to be too restrictive: Suppose that only two clone fragments in a clone group undergo a consistent change, while the other clone fragments in the
same group do not undergo any change, Saha’s definition will be not affirm such change as consistent, and their maintenance will be omitted, leading to inaccurate calculation of maintenance cost. In practice, it is apparent that – even changes only happen on two clone fragments – maintenance cost has been incurred. Therefore, in this work, we present a novel definition of consistent change to clone group that only requires at least two of code clones having consistent change, as follows,

**Definition 1 (Consistent Change):** A clone group $CG'$ in software version $j + 1$ possesses **consistent change** if there exists a pair of clones $CF_1'$, and $CF_2'$ in $CG'$ which are mappable to a pair of clones $CF_1$ and $CF_2$ in a clone group $CG$ in version $j$ such that modification of code pairs from $(CF_1, CF_2)$ to $(CF_1', CF_2')$ satisfies the following,

$$Simtext(CF_i, CF_i') < 1 \ \forall i \in \{1, 2\}$$

Our definition is constructed based on the intuition that appropriate changes to at least two clone fragments in a clone group (of possibly more than two clones) during its evolution requires attention, as it can incur consistency maintenance cost.

In the above definition, a clone group $CG'$ is **mappable** to another clone group $CG$ if there is a path linking from the nodes in $CG$ to the corresponding nodes in $CG'$ in the corresponding clone genealogy.

In general, majority of code clones are created via copy-and-paste operations by developers. Armed with clone genealogy technology, once a clone has been created, we can narrate how a clone group evolves through its life span. Recall that our prediction task aims to predict maintenance-consistency from any clone creating operation, we therefore formally define the notions of a clone creating instance and its consistency-requirement. Specifically,

**Definition 2 (Clone Creating Instance):** A clone group $CG$ in version $j$ is a **clone creating instance** if $CG$ is a root node in the clone genealogy ($CGE$).

In this definition, we identify a $CG$, which was created by a creation operation (such as copy-and-paste operation), as the root of a path in the clone genealogy $CGE$ that begins its evolution. This clone creating instance may link to some consistent change in its future evolution, and will either cause extra maintenance cost, or clone defects when failing to maintain their consistency in the group. Therefore, we perform prediction about this consistency of a clone creating instance, that can help developer mitigate clone consistent maintenance cost at the clone creating time. We call this clone **consistency-requirement**, defined as follows,

**Definition 3 (Clone Consistency-Requirement):** A clone creating instance $CG$ in the software version $j$ satisfies clone **consistency-requirement** if there is a different clone group $CG'$ in software version $k$, with $k > j$, such that (1) there is at least a pair of clones in $CG'$ that is mappable in clone genealogy to a pair of clones in $CG$, and (2) $CG'$ possesses “consistent change”. When $CG$ does not satisfy consistency-requirement, we say that it is consistency-requirement-free, or simply consistency-free.

Under this definition, if a consistent change occurs during clone creating instance’s evolution, this instance is deemed to require consistency maintenance in the future. Thus, we formalize our prediction task as follows:

**Research Problem:** Given a clone creating instance, determine if this instance meets consistency-requirement or it is consistency-free.

To accomplish this, we transform our prediction task into a classification problem of consistency-requirement, that will be addressed via machine learning methods.

### IV. METHODOLOGY

In this section, we detail our approach of consistency-requirement prediction for creating code clones.

#### A. OVERVIEW

We present our framework for performing prediction of consistency-requirement/free for any clone creating instance. The framework is shown in Figure 1. It includes three steps: Collection, representation and prediction.

The collection step aims to collect all clone creating instances from the subject software repository, so as to train the model of prediction with these collected labeled instances. Specifically, we use clone detection tool to detect all clones from all versions of software repository, and build clone genealogy through mapping between clone groups for all adjacent versions. Then, collection and labelling will be done through traversing clone genealogies and identifying consistent change in the evolution depicted in clone genealogy.

In representation step, each clone instance is represented by attributes consisting of all essential information about the creation surroundings. Specifically, we extract two sets of attributes to represent each clone creating instance: these two sets capture code and context properties respectively.

Finally, in the prediction step, we employ Bayesian network to predict clone consistency, and implement an Eclipse plug-in to aid developers performing prediction during development time. With this plug-in, developers can predict
whether a clone creating instance (occurred via creating a code clone) will need (or not need) consistent change in future. Specifically, if the model predicts that such a clone creating instance meets the consistency-requirement, the developer is advised to reconsider the need to create new code clones into his code. On the other hand, if the model predicts that such instance is free from meeting consistency-requirement, the developer will be informed as such, so that they can create the new code clones freely with more confidence.

In general, such prediction task always requires the knowledge of certain attributes, and here we employed the code attributes and context attributes for this purpose. Wang et al. use three set of attributes to predict clone consistency. We do likewise in this work but whittling down the historical attribute set and enrich the other two sets of attributes with more information. Through them, a Bayesian network will be trained to give us the probability of consistency requirement (and that of consistency free) of clone creating instance.

B. COLLECTION STEP
In order to predict consistency-requirement, we need to build and train models with training data collected from clone creating instances in software repository. Gratefully, all these clone instances can be obtained from clone genealogies of the software repository.

To construct a clone genealogy, we first detect all clone fragments and clone groups (Type-1, 2 and 3 clones) from each version of the software repository with NiCad [15]. Then, by pairing up (called mapping customarily) all these clones between every two consecutive versions of the software, we identify the evolutionary relationship for code clones.

We use a mapping algorithm, CRD-based Clone Group Mapping Algorithm [33], [34], to map all clone fragments and clone groups between two consecutive versions. Based on the mapped result, we obtain the clone genealogy for this software repository. For any one pair of mapped clone groups (from two consecutive versions), we determine whether the clone fragment of clone groups is modified by calculating the similarity between the mapped clone fragments in their clone group.

Finally, the clone group will be labeled for any consistent change it possesses (as specified in Definition 1). This collection task for clone creation can be accomplished by obtaining the root nodes of clone genealogies according to Definition 2. As this clone prediction task follows supervised learning model, we also need to associate each instance with its consistency-requirement or consistency-free indication (as defined in Definition 3).

For each clone creating instance, we need to identify the original code clones and the newly created clones. If such a clone group from root node of clone genealogy only have two clone fragments, the original code can be confirmed through checking if any code fragment from the previous version can be mapped to one of these two clone fragments. The mapped one is the original code clone, and the other one is the new creating clone. If no code can be mapped from the previous version, we choose one of these two fragments at random as the original code. If the clone group from the root node of the clone genealogy has more than two clone fragments, we split this clone group into more than one clone creating instances.²

C. REPRESENTATION STEP
We obtain two attribute sets from each clone creating instance; some of these attributes are also produced in the prediction work by Wang et al. [7]. In our work however, we enrich the code attributes and the context attributes to represent the original code and the created code clone respectively. Moreover, we remove the history attribute set used by Wang et al., and supply more detailed information in the remaining attribute sets.

The first set is code attribute set that captures the characteristics of the original code clone. These attributes describe the important lexical and syntax information from the code perspective. Some code attributes are similar to those described in the work by Wang et al.; these include the numbers of lines of code, the number of parameters, the number of call invocations, which can be referred to in [7]. In addition, we also collect “Halstead” metric and structural attributes. Halstead attributes are often used in software prediction. The structure attributes are pieces of information from the code syntax. These specific code attributes are detailed below:

- Number of Halstead: The number of Halstead for all the original code fragments in this clone creating instance. The Halstead includes the number of distinct operators, number of distinct operands, total number of operators, total number of operands.
- Number of Important Syntactic Constructs: For each important syntactic construct (such as if, while, etc.), we count the number of occurrences of the construct for the original code fragments in this clone creating instance.

The second set is context attribute set that captures the characteristics of the relationship between the original code and the associated code clone. Some of the context attributes are similar with Wang et al’s work, including locality of clones, file name similarity, method name similarity, sum of parameter similarities, and maximal parameter similarity, which can refer to [7]. In addition, we also extend the set to include code similarity, parameter type similarity and block information identification and other attributes. These specific context attributes are as follows:

- Clone Similarity: The similarity value between each pair of clones in the clone creating instance.
- Sum of Parameter Type Similarity (SPTS): Let $M_1$ and $M_2$ be the two clones with parameters $\text{Type}(P_1, P_2, \ldots, P_m)$ and $(Q_1, Q_2, \ldots, Q_n)$ respectively. We define SPTS

²Most of the clone groups occurring in the root node of the clone genealogy only have two clone fragments.
to be: \( \sum_{i=1}^{m} \sum_{j=1}^{n} \text{Sim}(P_i, Q_j) \). Here, \( \text{Sim} \) is a string similarity measure.

- “Is-same-block” Flag: This flag is raised if all the clones are enclosed within block statements of the same syntax.

**D. PREDICTION STEP**

Lastly, in order to apply Bayesian network for prediction, we need to build and train a predictive model in this step. For each subject software repository, we construct the training data set with the corresponding attributes of clone creating instances. These constructed data is then supplied to Bayesian network for the development of the model. The nodes in the Bayesian network model represent our attributes, and the edges denotes conditional dependencies among these attributes. With these instances, we finally generate well-trained predictors for our clone predictive tasks.

When a clone creating instance occurs, a developer can predict clone consistency-requirement with our well-trained predictor. Specifically, when the developer creates a piece of code clone, this creating instance will be detected and all attributes pertaining to this instance will be supplied to this predictor. This triggers the prediction for its consistency-requirement. When the developer creates new code clones by copy-and-paste operations, our tool can monitor these operations, and represent these creating instances by their corresponding attributes. According to the prediction results, we can inform the developer to take necessary measures for this new clones, either by accepting or rejecting the recommendation.

**V. IMPLEMENTATION**

To predict clone consistency-requirement during software development phase, we develop a prototype tool as an eclipse plug-in. In order to collect clone instances, our prototype constructs all clone genealogies for each of the software repositories based on the detection results of NiCad, and identifies the clone consistent change pattern for each pair of clone groups occurring between adjacent versions guided by the definitions. After that, our prototype extracts attribute sets for each clone instance, and generates the training dataset for machine-learning models with the attributes. Specifically, the tool extracts the code attribute set and context attribute set through analyzing the source code, and obtain the clone consistency-requirement label of clone instance through traversing the paths in the clone genealogies. Then, the Bayesian network predictor will be trained with these data through calling a third-party tool, WEKA [35], which provides all the APIs for the developer to make use of Bayesian network conveniently. When a clone creation occurs, our prototype captures this clone instance with the corresponding attributes; and feeds the results to the well-trained predictor, and obtains the prediction results for the developer’s consideration. In summary, the developer can now predict clone consistency when one clone creation occurs during software development.

Screenshots displayed in Figure 2 and Figure 3 depict our prototype tool in action. As shown in Figure 2, our prototype provides clone consistency prediction function by adding a new menu item of Clones in the eclipse menu bar. Before performing prediction, the developer should load the machine-learning model in advance. Our prototype provides three scenarios for loading the prediction model conveniently:

1) **Training scenario**: Given a software repository, this loads code clones (dataset) produced by the detection tool, and constructs the predictive model for the software repository.

2) **Data scenario**: This shortcut only loads a specified dataset for constructing the predictor, and the data can be obtained from other software repositories.

3) **Model scenario**: This straightforwardly loads a well-trained model for prediction; it thus saves the time for clone instances generation and the time for predictor construction.

When the predictive model is loaded successfully, our prototype can predict clone consistency-requirement by monitoring the copy-and-paste operations occurring during
TABLE 1. The statistics for clone creating instances.

| Repository     | Consistency-Free | Meeting-Consistency | Total |
|----------------|------------------|---------------------|-------|
| ArgouML        | 2574(77.07%)     | 766(22.93%)         | 3340  |
| JEdit          | 560(88.47%)      | 73(11.53%)          | 633   |
| JFreeChart     | 2013(59.80%)     | 1353(40.20%)        | 3366  |
| Tuxguitar      | 1016(71.10%)     | 413(28.90%)         | 1429  |

development. Fig. 3 is a screenshot of a clone creation operation that satisfies the criterion of clone consistency-requirement.

As shown, this prototype warns the developer that the operation will incur additional maintenance cost. As such, the developer can opt to reject this operation to avoid having new code clones and incurring additional maintenance costs in future evolution. On the other hand, our prototype tool will only predict the consistency when the developer performs a copy-and-paste operation. To help the developer maintain such consistency, we suggest to perform the clone consistency prediction when changing the code clones in a group (those interested readers can refer to [8] for discussion), and to perform consistent change automatically with the tool named JSync developed by Nguyen et al. [30].

VI. EXPERIMENTAL SETUP

Four open source repositories have been selected for our experiments. Their statistics are shown in Table 1. Here, we notice a large number of creating instances present in these repositories, ranging from 633 to 3366. The majority of them do not meet consistency-requirement, and these instances are shown in column 2, their counts ranged between 560 and 2574, with corresponding percentages ranging from 59.8% to 88.47%. On the other hand, there are a decent number of creating instances that meet the consistency-requirement (ranging from 73 to 1353 counts). This phenomenon gives the evidence that clone creating operation has truly become a common development technique, and majority of them will not introduce any consistent change in the future.

Similar to the work performed in [7], we divide our experiment into three parts to assess the effectiveness from three perspectives, including effectiveness experiment, attribute experiment, and cross-project experiment.

1) Effectiveness experiment: We assess the effectiveness of the prediction in this experiment. Here, we utilize all of the extracted attribute sets from each of the four repositories to train and test the model.

2) Attribute experiment: We assess the impact of these two contributing attribute sets on the prediction quality. This is accomplished through removing one set of attribute in the experiment.

3) Cross-project experiment: We assess the quality of the prediction on a project when the prediction model is built using data extracted from the other projects.

We employ WEKA—a flexible machine-learning tool—to construct and train the prediction model of Bayesian network. We also employ K2 algorithm to learn the network structure, and SimpleEstimator to build the conditional probability table of the network. The maximum number of parent nodes in Bayesian network is set to 3, so that it can accommodate the dependency among these attributes while not overly consuming too much memory and time.

For each creating instance, we calculate the probability of the consistency-requirement, both for meeting consistency and for being consistency-free. The probability values range between 0 and 1, and having a value closer to 1 indicates a high likelihood that the creating instance satisfies consistency-requirement criterion, and having a value closer to 0 means that the corresponding clone creating instance is likely to be consistency-requirement free.

Therefore, we set different thresholds for these two consistency-requirement in prediction. For instances that meet consistency-requirement, we set a range of higher threshold values from 0.5 to 1.0. This means that, when the predicted probability value of a clone creating instance is higher than a specific threshold value, we conclude that this instance satisfies clone consistency-requirement. By the same token, for predicting if a clone creating instance does meet consistency-free criteria, we set a lower threshold value from the range of values from 0 to 0.5. Analogously, when the value associated with the clone creating instance is lower than a specific threshold value, we conclude that the instance satisfies consistency-free requirement; i.e., this instance will not lead to any consistency maintenance issue during its evolution, and we recommend to the developer to consider introducing such clones into the current software.

To assess the quality of our model in predicting consistency-free, we compute the following three metrics:

- Recommendation Rate (RR): This percentage indicates the proportion of clone creating instances that our model predicts to be consistency-requirement free. It is computed as the ratio of the number of the instances recommended by the model to all clone creating instances tested.
- Precision (P): This assesses the accuracy of the model when it predicts that a clone creating instance under test to be consistency-free. It is computed as the ratio of the number of correct predictions of clone creating instances being consistency-free to the total number of predictions made by the model about clone creating instances being consistency-free.
- Recall (R): This assesses the effectiveness of the model in discovering all clone creating instances meeting the requirement of consistency-free. It is computed as the ratio of the number of correct predictions of clone creating instances being consistency-free to total number
of clone creating instances actually meeting the requirement of consistency-free.

On the other hand, for those clone-creating instances that meet consistency instances, they require consistency maintenance during evolution. We therefore warn developers to avoid creating these code clones. We compute the following three metrics for assessing the predictive quality of our model:

- **Warning Rate (WR):** This percentage indicates the number of clone creating instances that our model predicts to have met consistency-requirement, and thus warns developers to reject this copy-and-paste operation. It is computed as the ratio of the number of warning raised by the model to all clone creating instances tested.

- **Precision (P):** This assesses the accuracy of the model when it predicts that a clone creating instance under test meets the consistency-requirement. It is computed as the ratio of the number of correct predictions of clone creating instances meeting consistency-requirement to the number of predictions made by the model about clone creating instances meeting consistency-requirement.

- **Recall (R):** This assesses the effectiveness of the model in discovering all clone creating instances meeting consistency-requirement. It is computed as the ratio of the number of correct predictions of clone creating instances meeting consistency-requirement to total number of clone creating instances meeting consistency-requirement.

## VII. EXPERIMENTAL RESULTS

### A. EFFECTIVENESS EXPERIMENT

In this experiment, we utilize all attributes on four experimental projects, and employ cross-validation with 10-folds to train and test our prediction models. The results are shown in Figure 4 and Figure 5.

Figure 4 depicts the effectiveness of clone creating consistency-free on four projects. As can be seen from the table, our model achieves fairly well result in this prediction. For four projects at different thresholds, all precisions and recalls achieve a high level, that precisions ranging from 86.14% to 97.22% and recalls from 76.97% to 97.39%. While variation of threshold has some effect on both the precision and recall, it has stronger impact on recalls. This implies that the models can predict well, but can be further improved on its ability to recall. It also implies that developers can quite confidently rely on the recommendation given by the predictors. Meanwhile, our models have produced fairly reasonable recommend rates; it is around the percentage of creating instances meeting consistency-free (Table 1).

Figure 5 shows the effectiveness for meeting consistency on four projects. From this table, the models built for projects ArgoUML and jFreeChart offer effective prediction; the precision hovering around 90% and the recall around 80%. Both these projects perform fairly well, though that for Tuxguitar project is not as good. Unfortunately, our model does not predict well for jEdit project; this might be due to the small size of the creating instances that meeting consistency. While variation in threshold does affect these rates, it has stronger impact on precisions than recalls. This means that, the models created can predict well, but can be further improved on its ability to recall. In addition, the warning rates have been fairly reasonable for all the projects.

In summary, our approach produces models offer good prediction with high precision and recalls. It offers high confidence for developers to rely on the inference produced by our models.
B. ATTRIBUTE EXPERIMENT

In this work, we extract two sets of attributes that represent copied and pasted code clones in creating instance respectively. To assess the effect of each attribute set on the prediction power, we conduct attribute experiments that utilize only one attribute set with 10-folds cross-validation. The results are shown in Figure 6 and Figure 7. In these figures, we abbreviate Recommend Rate as “RR”, Warning Rate as “WR”, Precision as “P”, and Recall as “R”.

Figure 6 shows the significance of attribute set “Code” and “Context” respectively for meeting consistency-free, in comparison with using all attributes (“All”). From the “Code” lines, the predictive results are reasonably good, having only a slight drop in both precision and recall in comparison with “All” lines. This means that the attributes selected for code have positive effect both on precision and recall ability. From “Context” lines, although the precision has increased a little, the recall suffers compared with the case with all attribute set. This implies that the context attribute set has strong positive impact on precisions, and the code attribute has strong positive impact on recalls. Therefore, for consistency-free prediction, our attribute sets have their own positive impact on prediction, that context attributes have a great influence on precision, and code attributes on recall.

Figure 7 shows the significance of attribute sets for meeting consistency. For the ArgoUML and Tuxguitar projects, we obtain similar findings with consistency-free prediction, in comparison with using all attributes (“All”). From the “Code” lines, the predictive results are reasonably good, having only a slight drop in both precision and recall in comparison with “All” lines. This means that the attributes selected for code have positive effect both on precision and recall ability. From “Context” lines, although the precision has increased a little, the recall suffers compared with the case with all attribute set. This implies that the context attribute set has strong positive impact on precisions, and the code attribute has strong positive impact on recalls. Therefore, for consistency-free prediction, our attribute sets have their own positive impact on prediction, that context attributes have a great influence on precision, and code attributes on recall.

Figure 7 shows the significance of attribute sets for meeting consistency. For the ArgoUML and Tuxguitar projects, we obtain similar findings with consistency-free prediction, that extracted attributes have their own positive impact on the prediction. Specifically, context attributes have strong positive impact on precision, and code attributes have strong positive impact on recalls. For jEdit and jFreeChart projects, the effectiveness of prediction with only context attributes is better than all attribute sets. This implies that, when we predict from creating instances for meeting consistency, the attributes should be selected carefully for the best predictive power. Thus, the attribute sets have different positive impact for the different projects, and the context attributes may have more significant impact.

In summary, code and context attributes have played a positive role in predicting the consistency needs. We recommend keeping all attributes in constructing model, as some attributes might turn out to have significance for some yet-to-be-explored repositories.

C. CROSS-PROJECT EXPERIMENT

In the initial stage of software development, there may not have enough clone-creating instances from the software to develop the predictive model. We therefore construct a cross-project experiment, to investigate if models trained from other three projects can be used to predict the remaining fourth project. The results of cross-project prediction are shown in Figure 8 and Figure 9.

Figure 8 depicts the effectiveness in meeting consistency-free. Most of the precisions and recalls of the four projects suffer in comparison with the effectiveness experiment. Nevertheless, the predictive abilities of cross-project models are still acceptable, that precision ranging from 60.01% to 91.20% and recall ranging from 56.06% to 94.84%. The comparison shows that jEdit has the best predictive effect (higher accuracy), while jFreeChart has the worst prediction effect. The reason for the analysis may be due to the largest training set available for jEdit systems, thus having the most
comprehensive model training, and the jFreeChart being the opposite. If there is not sufficient data, model trained through cross-project can be employed for this clone consistency-free prediction. After several versions of evolution of the software, and with increasing amount of data from its own repository, it is recommended that the developer should retrain the model to predict consistency free of clone changes in the project.

Figure 9 depicts the results for meeting consistency-requirement. The prediction effect is quite low for the four projects: the predictive power of the four project is unacceptable in practice. The precisions for ArgoUML ranging from 7.93% to 20.66%, jEdit from 20.45% to 75%, jFreeChart from 33.33% to 52.59% and Tuxguitar from 44.31% to 51.85%. The recalls of these four projects hover at around 10%. The reason for this poor ability is that the number of instances requiring consistency is so small that the models are not trained well. Another reason is that prediction is also strongly dependent on the specific project itself, and may not...
be suitable for thus cross-way prediction. Therefore, in the meeting consistency-requirement prediction of cross-project, we do not recommend developer employed this the use of cross-validation approach to predict.

In summary, we can safely recommend that developers should perform consistency prediction with data from the same software. At the initial stage of development, we suggest that developers perform prediction of consistency-free with cross-project; and later, after several versions have been evolved, it’s advisable to retrain the model by progressively reducing cross-project data and increasing own data.

**D. COMPARISON**

As mentioned in the Introduction section, this work is inspired by the approach of Wang et al. [7] in predicting clone creating consistency. However, direct comparison between these two works is not practice – and not meaningful, due to the following reasons:
TABLE 2. The statistics for clone creating instances.

| Project            | Consistency-Free | Meeting-Consistency | Total  |
|--------------------|------------------|---------------------|--------|
| jFreeChart(ConQAT) | 1243(65.04%)     | 668(34.96%)         | 1911   |
| jFreeChart(NiCad)  | 2013(59.80%)     | 1353(40.20%)        | 3366   |
| Tuxguitar(ConQAT)  | 538(70.52%)      | 225(29.48%)         | 763    |
| Tuxguitar(NiCad)   | 1016(71.10%)     | 413(28.90%)         | 1429   |

TABLE 3. The comparing effectiveness for the predictors.

| Projects             | Average-Precision | Average-Recall |
|----------------------|-------------------|----------------|
| jFreeChart(ConQAT)   | 89.20%            | 89.30%         |
| jFreeChart(NiCad)    | 88.30%            | 88.20%         |
| Tuxguitar(ConQAT)    | 82.70%            | 83.10%         |
| Tuxguitar(NiCad)     | 83.10%            | 83.60%         |

1) The definitions of consistent change (Definition 1) are different. As mentioned in Section III, we believe that maintenance cost would still be incurred even if two out of many clone fragments within a clone group have been modified consistently. Therefore, the number of clone groups meeting the consistency-change requirements will be different in these two works. (The number of consistent-change clone groups in our work will be more than that of Wang et al.)

2) Operationally, Wang et al. conducted empirical study on four projects including two from Microsoft Inc and two open source projects of iFreeChart and Tuxguitar. We are unable to obtain the repositories for the first two projects.

3) Operationally, the clone genealogy we develop is different from that developed by Wang et al. Specifically, we develop genealogy from all the released versions in a repository, whereas Wang et al. developed genealogy from all the commits done to their collected (but not all) versions in the same repository.

Some concerns have been raised about the significance of using different clone detection tools in these two works; i.e., using ConQAT (by Wang et al.) and NiCad (by our work). To this end, we re-conducted the experiments by replacing NiCad with ConQAT. The statistics about the number of clone instances possessing consistency-free or meeting consistency using these two tools on two project are collected and shown to be similar, as depicted in Table 2:

Table 3 shows the comparison between the effectiveness of the our approach using two different clone detection tools on these two projects. Here, we employ average-precision and average-recall as the metrics for evaluation. These average all the precision numbers and recall numbers for clone creating instances of meeting consistency and the instances of consistency-free. As shown in this table, regardless of whether ConQAT or NiCad is used as detection tool, their predictive powers remain closely similar, and respectable.

E. DISCUSSION

We evaluate the effectiveness of the prediction models that are constructed based on Bayesian network both for clone consistency-requirement and consistency-free.

Based on the results of our effectiveness experiments, our approach possesses effective prediction ability, on both consistency-requirement and consistency-free prediction, when the models are trained with full attributes.

The attribute set experiment investigates the models’ effectiveness by only employing single attribute set. It reveals the significance of the contributions by each attribute set, showing that each has its own positive impact on the quality of the predictive model. In addition, we also encourage developers to add in other attributes to further sharpen the predictability of the constructed model.

The cross-project experiment shows that prediction model effectiveness is highly dependent on the specific characteristics of the repository on which the model is to be applied. The results also show that a “universal” predictive model built with data drawing from other software repositories cannot effectively perform consistency-requirement prediction, whereas it has acceptable degree of ability for consistency-free prediction. Therefore, developers will need to create different models on the consistency-free prediction for different repositories, that is certainly not entirely desirable, and we feel that further investigation is required here.

VIII. THREATS TO THE VALIDITY

The first threat to the validity of this empirical research concerns the construct validity that whether the metrics used in the construction and the evaluation are appropriate. There are two potential threats here:

1) Our assumption that the root nodes in the clone genealogy are obtained from copy-and-paste operations; this may not be accurate. Regardless of the validity of this assumption, we note that these are undoubtedly clone-creating instances, since clones are indeed detected for the first time in software evolution process. We are however not able to confirm that these clones are created by the copy-and-paste operations, but can only point out the prevalent use of copy-and-paste in software development.

2) We remain silent about how to determine the maintenance cost incurred from consistency-check of clones, and assume that it is uniform for all clone-creating instances. Nevertheless, we assume that a consistent change in at least two clone fragments in a group is the basic criterion for extra maintenance cost in clone evolution. This criterion may not properly reveal the real cost of software maintenance, but it indicates some minimal costs incurred. In other words, we believe that such consistent change occurring in a clone group indeed increases the risk of some clone maintenance cost and consistent-defect.
Next, threats to validity of conclusion concerns with the sufficiency of clone instances in software repository in order to train the prediction model. Specifically, the repository supplies data to machine learning methods, which require adequate data to train the prediction model well. Therefore, software repository should ideally contain sufficient number of revisions or have adequate occurrences of clone instances. In general, we acknowledge that our approach may not be suitably applicable in predicting clone consistency-requirement for new repository or for repositories with insufficient data. Furthermore, to mitigate such a threat in general, we have shown earlier a use case of cross-project prediction which trains a predictive model using matured projects from other repositories. Under this situation, we recommend developers to perform this prediction only for consistency-free requirements.

For other validity threat, external validity concerns the impact from different types of code clones. We do not consider these variations because of the following reasons: 1) The types of code clones may shift from Type-1 to Type-2 or Type-3, which can be caused by changes occurred on code clones by the developers for some reason. 2) The quantity of these 4 Types of code clones for each repository are very different, which affect the outcome of the prediction. The prediction model needs sufficient clone instances to train. However, the existing clone detection tools are not capable of identifying enough code clones of these varying types. Nevertheless, the empirical study on the different types of code clones is worth observing, and we will try to make different predictions on the different kinds of type clones, which is certain an interesting work to be investigated in future.

IX. CONCLUSION AND FUTURE WORK

Ensuring consistent change of code clones during software evolution can incur extra maintenance cost to the software development process. This additional burden can be mitigated if developers can be prompted to consider such situation when clones are created (mainly through copy-and-paste operation) in the beginning. This work takes this proactive approach, and predicts at clone-creating time whether a clone-creating instance can cause consistency-requirement or being consistency-free. To this end, we build a prediction model based on Bayesian network. We feed the network with information about clones present in each clone-creating instance, represented by two sets of attributes – code attributes and context attributes, respectively. In this paper, we have described how such a model is built, and conducted experiments on four open source projects to investigate three perspectives of the effectiveness of prediction models. The experimental results show that the model has been effective in performing prediction, with high precision and recall rates for these repositories. In addition, both the two sets of attributes chosen have positive impact on the predictions, and contribute significantly to both precision and recall ability. Lastly, through cross-project experiment, we show that the prediction model can be employed for consistency-free prediction. Therefore, this paper recommends that the system needs to predict the consistency of demand, the preferred use of their own data for training, and according to different systems to select different measures to meet the needs of developers.

To follow up, we intend to build a general cross-project predictor to enhance the predictability in the presence of new software repository by introducing new attributes. We believe that this holistic approach can be a solid boost to software maintainability to improve software quality. As developers can introduce code clones more freely and more confidence with the awareness of when consistency-requirement of a new clone group need to be investigated, avoiding the potential cost of consistency maintenance.

REFERENCES

[1] D. Chatterji, J. C. Carver, and N. A. Kraft, “Code clones and developer behavior: Results of two surveys of the clone research community,” Empirical Softw. Eng., vol. 21, no. 4, pp. 1476–1508, Aug. 2016.
[2] M. Kim, V. Sazawal, D. Notkin, and G. Murphy, “An empirical study of code clone genealogies,” in Proc. ACM SIGSOFT Softw. Eng. Notes, 2005, vol. 30, no. 5, pp. 187–196.
[3] M. Mondal, M. S. Rahman, C. K. Roy, and K. A. Schneider, “Is cloned code really stable?” Empirical Softw. Eng., vol. 23, no. 2, pp. 693–770, Apr. 2018.
[4] R. K. Saha, C. K. Roy, and K. A. Schneider, “An automatic framework for extracting and classifying near-miss clone genealogies,” in Proc. 27th IEEE Int. Conf. Softw. Maintenance (ICSM), Sep. 2011, pp. 293–302.
[5] M. Mondal, C. K. Roy, and K. A. Schneider, “Does cloned code increase maintenance effort?” in Proc. IEEE 11th Int. Workshop Softw. Clones (IWSC), Feb. 2017, pp. 1–7.
[6] S. Wagner, A. Abdulhakeel, K. Kaya, and A. Paar, “On the relationship of inconsistent software clones and faults: An empirical study,” in Proc. IEEE 23rd Int. Conf. Softw. Anal., Evol., Reeng. (SANER), vol. 1, Mar. 2016, pp. 79–89.
[7] X. Wang, Y. Dang, L. Zhang, D. Zhang, E. Lan, and H. Mei, “Predicting consistency-maintenance requirement of code clones via copy-and-paste time,” IEEE Trans. Softw. Eng., vol. 40, no. 8, pp. 773–794, Aug. 2014.
[8] F. Zhang, S.-C. Khoo, and X. Su, “Predicting change consistency in a clone group,” J. Syst. Softw., vol. 134, pp. 105–119, Dec. 2017.
[9] N. Friedman, D. Geiger, and M. Goldszmidt, “Bayesian network classifiers,” Mach. Learn., vol. 29, no. 2, pp. 131–163, Nov. 1997.
[10] M. Fowler and K. Beck, Refactoring: Improving the Design of Existing Code. Reading, MA, USA: Addison-Wesley, 1999.
[11] S. Wagner, A. Abdulhakeel, K. Kaya, and A. Paar, “On the relationship of inconsistent software clones and faults: An empirical study,” in Proc. IEEE 11th Int. Workshop Softw. Clones (IWSC), Feb. 2017, pp. 1–7.
[12] F. Zhang, S.-C. Khoo, and X. Su, “Predicting consistent clone change,” in Proc. IEEE 27th Int. Symp. Softw. Rel. Eng. (ISSRE), Oct. 2016, pp. 353–364.
[13] M. Mondal, C. K. Roy, and K. A. Schneider, “Code clones and developer behavior: Results of two surveys of the clone research community,” Empirical Softw. Eng., vol. 21, no. 4, pp. 1476–1508, Aug. 2016.
[14] M. Mondal, C. K. Roy, and K. A. Schneider, “Is cloned code really stable?” Empirical Softw. Eng., vol. 23, no. 2, pp. 693–770, Apr. 2018.
[15] C. K. Roy and J. R. Cordy, “NICAD: A comparative study on different clone types,” J. Syst. Softw., vol. 144, pp. 41–59, Oct. 2018.
[16] R. K. Saha, C. K. Roy, and K. A. Schneider, “Clone propagation tendency of code clones,” in Proc. 5th Int. Workshop Softw. Clones (IWSC), 2011, pp. 28–33.
[17] F. Zhang, S.-C. Khoo, and X. Su, “Predicting consistent clone change,” in Proc. IEEE 27th Int. Symp. Softw. Rel. Eng. (ISSRE), Oct. 2016, pp. 353–364.
F. Zhang et al.: Improving Maintenance-Consistency Prediction During Code Clone Creation

[19] D. Rattan, R. Bhatia, and M. Singh, “Software clone detection: A systematic review,” Inf. Softw. Technol., vol. 55, no. 7, pp. 1165–1199, Jul. 2013.

[20] M. White, M. Tufano, C. Vendome, and D. Poshyvanyk, “Deep learning code fragments for code clone detection,” in Proc. 31st IEEE/ACM Int. Conf. Automated Softw. Eng. (ASE), New York, NY, USA: ACM, 2016, pp. 87–98, doi: 10.1145/2970276.2970326.

[21] V. Saini, F. Farmahinifarahani, Y. Lu, P. Baldi, and C. V. Lopes, “Oreo: Detection of clones in the twilight zone,” in Proc. 26th ACM Joint Meeting Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng. (ESEC/FSE), 2018, pp. 354–365.

[22] L. Li, H. Feng, W. Zhuang, N. Meng, and B. Ryder, “CC Learner: A deep learning-based clone detection approach,” in Proc. IEEE Int. Conf. Softw. Maintenance Evol. (ICSME), Sep. 2017, pp. 249–260.

[23] H. Wei and M. Li, “Supervised deep features for software functional clone detection by exploiting lexical and syntactical information in source code,” in Proc. 26th Int. Joint Conf. Artif. Intell., Aug. 2017, pp. 3034–3040.

[24] A. Narayanan, M. Chandramohan, L. Chen, Y. Liu, and S. Sami-nathan, "Subgraph2vec: Learning distributed representations of rooted sub-graphs from large graphs," 2016, arXiv:1606.08928. [Online]. Available: http://arxiv.org/abs/1606.08928

[25] A. Sheneamer, S. Roy, and J. Kalita, “A detection framework for semantic code clones and obfuscated code,” Expert Syst. Appl., vol. 97, pp. 405–420, May 2018.

[26] Y. Higo, S. Kasumoto, and K. Inoue, “A metric-based approach to identifying refactoring opportunities for merging code clones in a Java software system,” J. Softw. Maintenance Evol., Res. Pract., vol. 20, no. 6, pp. 435–461, Nov. 2008.

[27] G. P. Krishnan and N. Tsantalis, “Unification and refactoring of clones,” in Proc. Softw. Evol. Week-IEEE Conf. Softw. Maintenance, Reeng., Reverse Eng. (CSMR-WCRE), Feb. 2014, pp. 104–113.

[28] N. Tsantalis, D. Mazinanian, and G. P. Krishnan, “Assessing the refactorability of software clones,” IEEE Trans. Softw. Eng., vol. 41, no. 11, pp. 1055–1090, Nov. 2015.

[29] C. K. Roy, M. F. Zibran, and R. Koschke, “The vision of software clone management: Past, present, and future (Keynote paper),” in Proc. Softw. Evol. Week-IEEE Conf. Softw. Maintenance, Reeng., Reverse Eng. (CSMR-WCRE), Feb. 2014, pp. 18–33.

[30] A. Rahmati and I. Kurban, “A software clone detection algorithm using graph structure,” in Proc. 31st IEEE/ACM Int. Conf. Automated Softw. Eng. (ASE), New York, NY, USA: ACM, 2016, pp. 87–98, doi: 10.1145/2970276.2970326.

[31] M. White, M. Tufano, C. Vendome, and D. Poshyvanyk, “Deep learning code fragments for code clone detection,” in Proc. 31st IEEE/ACM Int. Conf. Automated Softw. Eng. (ASE), New York, NY, USA: ACM, 2016, pp. 87–98, doi: 10.1145/2970276.2970326.

[32] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten, “The WEKA data mining software: An update,” ACM SIGKDD Explor. Newsletter., vol. 11, no. 1, pp. 10–18, 2009.