Predicting Consistency-Maintenance Requirement of Code Clones at Copy-and-Paste Time

Xiaoyin Wang*, *Member, IEEE*, Yingnong Dang, Lu Zhang, Dongmei Zhang, Erica Lan, and Hong Mei, *Senior Member, IEEE*

Abstract—Code clones have always been a double edged sword in software development. On one hand, it is a very convenient way to reuse existing code, and to save coding effort. On the other hand, since developers may need to ensure consistency among cloned code segments, code clones can lead to extra maintenance effort and even bugs. Recently studies on the evolution of code clones show that only some of the code clones experience consistent changes during their evolution history. Therefore, if we can accurately predict whether a code clone will experience consistent changes, we will be able to provide useful recommendations to developers on leveraging the convenience of some code cloning operations, while avoiding other code cloning operations to reduce future consistency maintenance effort. In this paper, we define a code cloning operation as *consistency-maintenance-required* if its generated code clones experience consistent changes in the software evolution history, and we propose a novel approach that automatically predicts whether a code cloning operation requires consistency maintenance at the time point of performing copy-and-paste operations. Our insight is that whether a code cloning operation requires consistency maintenance may relate to the characteristics of the code to be cloned and the characteristics of its context. Based on a number of attributes extracted from the cloned code and the context of the code cloning operation, we use Bayesian Networks, a machine-learning technique, to predict whether an intended code cloning operation requires consistency maintenance. We evaluated our approach on four subjects — two large-scale Microsoft software projects, and two popular open-source software projects — under two usage scenarios: 1) recommend developers to perform only the cloning operations predicted to be very likely to be consistency-maintenance-free, and 2) recommend developers to perform all cloning operations unless they are predicted very likely to be consistency-maintenance-required. In the first scenario, our approach is able to recommend developers to perform more than 50% cloning operations with a precision of at least 94% in the four subjects. In the second scenario, our approach is able to avoid 37% to 72% consistency-maintenance-required code clones by warning developers on only 13% to 40% code clones, in the four subjects.

Index Terms—Code cloning, Consistency Maintenance, Programming aid

1 Introduction

In software development, developers frequently reuse code to reduce development effort. To reuse a piece of code, a developer may choose to wrap the reused code into a module (e.g., a method), and invoke the module wherever the code is required. Alternatively, the developer may also copy the code and paste it to the place where he or she wants to reuse the code with some revisions if necessary. In most cases, it is more convenient for developers to perform copy-and-paste to reuse existing code. Researchers have found that, developers often do not have enough time to wrap reused code to modules due to the schedule pressure of coming software releases [39]. Additionally, wrapping reused code may require developers to change the code written by other people / organization, which is technically challenging and needs extra management costs [14].

As a result, in the code base of modern software projects, code clones resulted from copy-and-paste based code reuse are very common. For example, it is reported that, in the code base of Eclipse-JDT, more than 20% of the code is involved in code clones or near-clones [37]. These code clones, while bringing convenience to the developers, also cause consistency maintenance. In many cases, developers need to maintain the consistency among the clone segments when they want to revise one clone segment[1]. Otherwise,

- *Xiaoyin Wang is with the Key Laboratory of High Confidence Software Technologies (Peking University), Ministry of Education, and with the Department of Computer Science, University of Texas at San Antonio. E-mail: Xiaoyin.Wang@utsa.edu*
- *Lu Zhang, and Hong Mei are with the Key Laboratory of High Confidence Software Technologies (Peking University), Ministry of Education, and Institute of Software, School of Electronics Engineering and Computer Science, Peking University, Beijing, 100871, P.R. China E-mail: {zhanglu,meih}@sei.pku.edu.cn*
- *Yingnong Dang and Dongmei Zhang are with Microsoft Research Asia, Beijing, 100871, China. E-mail: Dang.Yingnong, Zhang.Dongmei@microsoft.com*
- *Erica Lan is with Microsoft Corporation, One Microsoft Way, Redmond, WA, USA. E-mail: erical@microsoft.com*

**A large portion of the work was done when the author was a PhD student in Peking University and working as an intern in Microsoft Research*

1. In this paper, when several pieces of code are clones of each other, we use the term *clone segment* to represent one of these pieces of code

their revisions may become bug-inducing [26] [36].

However, according to recent studies, a large portion of code clones in a software project may not be detrimental to software evolution and maintenance [38] [21]. In a recent study by Göde and Koschke [21], it is reported that less than half of the code clones may change during their life cycle. Furthermore, only about half of the changes happening on code clones are consistent changes (i.e., simultaneous changes of more than one segment in a code clone), while the other changes are inconsistent changes (i.e., changes to only one clone segment without touching other clone segments). Typically, when performing inconsistent changes[2], developers do not need to spend extra time on considering other clone segments. Thus, extra maintenance effort is needed only when consistent changes are required.

Based on whether the resulting code clone requires consistency maintenance, we define two categories of cloning operations in this paper. The first category of cloning operations, referred to as *consistency-maintenance-free cloning operations*, includes cloning operations whose resulting code clones never change or always change inconsistently. The second category of cloning operations, referred to as *consistency-maintenance-required cloning operations*, includes cloning operations whose resulting code clones need to be changed consistently. For consistency-maintenance-free cloning operations, no extra maintenance of consistency is required later, so that developers can perform copy-and-paste and benefit from the convenience almost for free; while for consistency-maintenance-required cloning operations, consistent changes will incur extra maintenance effort and even lead to code defects if consistency is not well maintained. If a developer is reminded at the copy-and-paste time that the undergoing cloning operation may cause requirement for consistency maintenance, he or she is able to immediately consider alternative ways (e.g., wrapping the clone code into a module) for cloning.

It should be noted that, there can be many other factors (such as development schedule, general design of software, difficulty to do refactoring) that affect the developers' decision on whether to perform a code cloning operation. However, we believe that potential consistency-maintenance requirement is at least one of the major factors because convenience for future maintenance is one of the major goals of good software design. Therefore, assistance in understanding the potential consistency-maintenance requirement of an intended cloning operation may help developers make a better choice on how to reuse certain existing code.

Due to various reasons (e.g., lack of expertise or lack of supporting information), developers cannot always wisely determine whether a cloning operation is likely to be consistency-maintenance-required. Previous research effort [26] has found a number of software defects caused by incorrectly performing consistent changes on code clones. Furthermore, the statistics on our evaluation subjects (See Table 5 and 14) show that the developers of the subjects allow a non-trivial number of consistency-maintenance-required cloning operations, which account for about 10% to 15% of all cloning operations allowed.

In this paper, we propose an automatic approach to predicting whether intended cloning operations are consistency-maintenance-required. The intuition of our work is that whether a code cloning operation requires consistency maintenance is related to certain characteristics of the code to be cloned and the context of the code cloning operation. Therefore, we extract a number of attributes from the cloned code and the context of the code cloning operation. Then, we use Bayesian Networks, a machine-learning technique, to predict whether an intended code cloning operation requires consistency maintenance. Specifically, we use three groups of attributes in our prediction: historical attributes that describe the change history related to the code to be cloned, code attributes that describe syntactical characteristics of the code to be cloned, and destination attributes that characterize the target place (i.e., the place where the developer intends to paste the code). For a software project with history, we train a machine-learning model from the code-cloning operations happened in the history of the software project. For newly launched projects without history data, we train the machine-learning model from the history of other projects.

To evaluate our approach, we applied our approach on four subjects. The four subjects include two large-scale software projects from Microsoft, and two popular open-source software projects from SourceForge [3]. When applying our approach, we considered two possible usage scenarios: the conservative scenario for cautious developers who want to perform cloning operations only when they are sure that the cloning operations would be unlikely to cause consistency-maintenance requirement, and the aggressive scenario for radical developers, who want to avoid a significant proportion of consistency-maintenance-required cloning operations while still being able to perform most intended cloning operations. For the first scenario, our approach recommends developers to perform only cloning operations predicted to be very likely to be consistency-maintenance-free; while for the second scenario, out approach warns developers only on cloning operations predicted to be very likely

2. Here we use the term "divergent changes" in the rest of the paper for intentional inconsistent changes. In reality, there may exist unintentional inconsistent changes that lead to bugs, but studies [22] show that such cases are relatively rare for well tested software systems.

3. http://sourceforge.net/

to be consistency-maintenance-required. Our evaluation results demonstrate that 1) with precision higher than 94.0%, our approach is able to predict 52% to 60% of cloning operations as consistency-maintenance-free for the four projects in the conservative scenario, and 2) by predicting only 13% to 40% of cloning operations as consistency-maintenance-required in the four projects, our approach is able to avoid 37% to 72% of consistency-maintenance-required cloning operations in the aggressive scenario.

The main contributions of this paper are as below:

- We identify a number of attributes that may relate to the potential consistency-maintenance requirement of code cloning operations, and demonstrate that it is possible to differentiate between consistency-maintenance-required code cloning operations and consistency-maintenance-free cloning operations.
- We propose an automatic approach to predict whether cloning operations require consistency maintenance based on a Bayesian Network classification model trained with software version history data.
- We implement our approach as two prototypes. One for C# projects in Microsoft, and the other for Java projects based on subversion.
- We evaluate our approach for two usage scenarios on four software projects from both industry and open source community.

This paper is an extended and revised version of our previous conference paper [44]. The main difference between this paper and our previous conference paper are as follows. First, besides the implementation of our approach for C# software projects based on the Microsoft version system, we re-implement our approach for subversion-based Java software projects. Second, on top of two industry software projects from Microsoft, we further evaluate our approach on two open source software projects. Third, we present more details about our approach, implementation and evaluation, and we do a more thorough discussion of the related research efforts.

We organize the rest of this paper as follows. In Section 2, we provide two examples of consistency-maintenance-required and consistency-maintenance-free cloning operations. In Section 3, we present our approach in detail. In Section 4, we introduce the implementation of our two prototypes and their differences. In Section 5, we present an evaluation of our approach using industrial software projects from Microsoft and open source software projects. We discuss some related issues in section 6. We introduce related work in Section 7 and present future works in Section 8. Finally, we summarize our contributions and conclusions of studies in Section 9.

2 EXAMPLES

In this section, we illustrate the research problem of this paper with two examples. Both examples are cloning operations from the development history of the first project (i.e., XProj[4], from Microsoft) used in our evaluation. The first example depicts a typical consistency-maintenance-required cloning operation and the second example depicts a typical consistency-maintenance-free cloning operation. From these two examples, we can discover some clues about how consistency-maintenance-required and consistency-maintenance-free cloning operations are different from each other. It should be noted that, for brevity, some minor part of the code (e.g., empty lines, comments) are trimmed.

2.1 A Consistency-Maintenance-Required Clone

In the first cloning operation, the developer copied 23 lines of code (i.e., Code Snippet 1) and pasted the copied code into the same method with slight revision (i.e., variables names and constants are changed from PPE-related to PROD-related as highlighted in Code Snippet 2). After the copy-and-paste operation, the two clone segments experienced four consistent changes, which happened 2, 12, 32, and 33 months later, respectively.

1) In the first change, the developers changed the variable `cockpitServers`, `cockpitPorts`, and `collectionDirs` (accessed in Lines 2, 4, and 6) to field variables. Thus the name of the variables are changed according to naming rules (e.g., `cockpitServers` is changed to `m_cockpitServers`). Since the two code snippets are in the same method, and access all the three local variables. The variable access in Lines 2, 4, and 6 of both code snippets were changed accordingly.
2) In the second change, the developers inserted another invocation of the `Config.GetParameter` method after Line 3 to fetch an extra configuration variable (`m_cockpitEnvs`).
3) In the third change, to obey new naming rules, the developers added a prefix "StaticRank3" to the string constant "PPE" in code snippet 1, and the string constant "PROD" in code snippet 2.
4) In the fourth change, the developers changed the `LogLevel.Warning` in Line 21 of both code snippets to `LogLevel.Error`.

From this example, we can identify two factors for consistent changes. First, the clone segments interact with the same configuration and static fields so that changes in the configuration or static fields may impact both clone segments simultaneously. Second, the copy-and-paste is local so that the copied and pasted

4. According to Microsoft regulations, we are unable to disclose the names and the application domains of the two projects.

code pieces share many local variables which need to be changed consistently in both clone segments.

Code Snippet 1:
```
try{
  cockpitServers["PPE"]=Config.GetParameter(c_iqmFile,
    "PPE","cockpitServer",c_ppeCockpitServer);
  cockpitPorts["PPE"]=Config.GetIntParameter(
    c_iqmFile,"PPE","cockpitPort",c_ppeCockpitPort);
  collectionDirs["PPE"]=Config.GetParameter(c_iqmFile,
    "PPE","collectionDir",c_ppeCollectionDir);
  if ( Config.GetIntParameter(
    c_iqmFile,"PPE","rankDataAvailable",
    c_ppeRankDataAvailable ) == 1 ){
    m_numCollections++;
    rankDataAvailable["PPE"] = true;
  }
  if ( Config.GetIntParameter(
    c_iqmFile,"PPE","crawlDataAvailable",
    c_ppeCrawlDataAvailable ) == 1 ){
    m_numCollections++;
    crawlDataAvailable["PPE"] = true;
  }
}catch{
  Logger.Log( LogID.IQM,LogLevel.Warning, "Unable
    to get cockpitServer or cockpitPort for PPE");
}
```
Code Snippet 2:
```
try{
 cockpitServers["PROD"]=Config.GetParameter(c_iqmFile,
   "PROD","cockpitServer",c_productionCockpitServer);
 cockpitPorts["PROD"]=Config.GetIntParameter(c_iqmFile,
   "PROD","cockpitPort",c_productionCockpitPort);
 collectionDirs["PROD"]=Config.GetParameter(c_iqmFile,
   "PROD","collectionDir",c_productionCollectionDir);
 if ( Config.GetIntParameter(
   c_iqmFile,"PROD","rankDataAvailable",
   c_productionRankDataAvailable ) == 1 ){
   m_numCollections++;
   rankDataAvailable["PROD"] = true;
}
  if ( Config.GetIntParameter(
    c_iqmFile,"PROD","crawlDataAvailable",
    c_productionCrawlDataAvailable) == 1 ){
    m_numCollections++;
    crawlDataAvailable["PROD"] = true;
  }
}catch{
  Logger.Log( LogID.IQM,LogLevel.Warning, "Unable to
     get cockpitServer or cockpitPort for production");
}
```

2.2 A Consistency-Maintenance-Free Clone

In the second cloning operation, the developer copied the code depicted below and pasted it into another method in another file without change. The clone group generated by the copy-and-paste operation remained unchanged for 3 years until the module containing this clone group was completely removed. From this example, we may identify the following factors for consistency-maintenance-free cloning operations. First, the copied code piece is pasted to another method in another file. Therefore, the resulting clone segments do not share local methods and variables. Second, the code piece contains only library function calls so that it has relatively less dependence on other modules and is unlikely to be affected by a change in other modules. This piece of code still has some dependencies on the other part of the project. For example, `sSource` is a function parameter, which depends on its calling modules. But since the two code snippets are in different methods, the calling modules may also be different. So even if the code need to be changed due to this dependency, the change is more likely to be an inconsistent one.

```
if (c >= sSource.Length ||
  (c == sSource.Length - 1 && sSource[c] == '&')){
   break;
}
if (sSource[c] == '&'){
   try{
     if (sSource[c+1] == 'q' && sSource[c+2] == '=' &&
       sTag.ToLower() == "&q="){
       c += 3;
     }else{
       break;
     }
   }catch (Exception e){
     Console.WriteLine(e.Message);
   }
}
```

2.3 Analysis

From the two examples above, we can see that there do exist some clues in cloning operations that relate to their consistency-maintenance requirement. For example, more dependence of the code to be cloned on other code may indicate more likelihood of being consistency-maintenance-required. Therefore, it should be feasible for us to determine the consistency-maintenance requirement of an intended cloning operation at the copy-and-paste stage. However, the preceding two examples also demonstrate that it might be difficult to obtain simple discriminative rules based on the clues. Therefore, it is a reasonable choice to leverage a machine-learning based technique to learn an effective predictor from the evolution histories of existing code clones, because version histories of existing software projects record a large number of cloning operations and the evolution of the resulting code clones over time.

3 Approach

In this section, we first introduce some background knowledge about the Bayesian Network classification model. Then, we give the overview of our approach. After that, we present the considered attributes in our specific classification model, and how we extract attributes for historical code cloning operations to train the model. Finally, we present the two scenarios that our prediction approach can be applied to help developers.

3.1 Background

Bayesian Networks are a mathematical model developed in late 1980s [2], which are used to predict the probability of an event based on the happening of other observable events. Formally, a Bayesian Network is a directed acyclic graph, in which each node represents an event, and the weight on the edge from node A to node B represents the conditional probability of event B provided that event A happens. Therefore, given a Bayesian Network, it is easy to calculate the probability of an event based on the parent nodes of the event's corresponding node, if all the parent nodes are observable.

To build a Bayesian Network, one needs to decide the nodes, the structure, and the weights on the edges in the Bayesian Network. Typically, the nodes correspond to the observable events that relate to the prediction goal. After deciding the nodes in the Bayesian Network, the structure and the weights of the Bayesian Network can be learned from a number of training instances. A training instance is a vector that indicates whether each observable event happens. The typical algorithm for learning the structure of a Bayesian Network is the K2 algorithm [4], which tries to maximize the maximal probability of the training instances for all possible weights. The typical algorithm for learning the weights of a Bayesian Network is the maximum likelihood approach [4], which tries to maximize the probability of the training instances.

In this paper, we rely on the implementation of the learning algorithms in Weka [19] to construct the Bayesian Network from a number of cloning operations whose attributes and consistency-maintenance status are already known. Besides Bayesian Networks, there are actually many other machine learning approaches, such as Decision Trees [41], Support Vector Machines (SVM) [40]. We choose to use Bayesian Networks because it has off-the-shelf implementations, and it is less likely to get over-fitting[5] [42]. It should be noted that our approach is general and may leverage any classification algorithm. It is not known whether Bayesian Network is the best classification algorithm for this specific problem. So it is possible that other classification algorithms may further enhance the effectiveness our approach, and we plan to study the effectiveness of other classification algorithms in the future.

3.2 Overview

The basic idea of our approach is to transform the problem of predicting whether intended cloning operations require consistency maintenance into a problem of learning a classification model from existing data of performed cloning operations. In particular, we adopt Bayesian Networks. Based on this idea, our approach mainly includes two phases. The first phase is the predictor construction phase, in which we build a Bayes-Network classification model as a predictor from a number of copy-and-paste operations whose attributes and consistency-maintenance status are already known. The second phase is the predictor application phase, in which the developers use the predictor to predict whether the newly made copy-and-paste operations require consistency maintenance.

5. In the area of machine learning, the term over-fitting describe a case in which the trained classification model is extremely optimized for the training set, so that it will not perform well on a testing set that is different from the training set to some extent.

The predictor construction phase of our approach includes three steps. First, to acquire a sufficient number of training copy-and-paste operations, we analyze the original generation of code clones in historical versions of software projects. Second, to determine whether the acquired historical copy-and-paste operations require consistency maintenance, we trace the evolution history of the code clones generated by the acquired copy-and-paste operations. Third, we extract the attributes of each copy-and-paste operation.

3.3 Considered Attributes

To construct the Bayesian Network classification model for prediction, we use a set of attributes that may be related to whether intended cloning operations require consistency maintenance. Specifically, we use 21 attributes in total, and these attributes fall into three categories: history attributes, code attributes, and destination attributes.

3.3.1 History Attributes

When a developer performs copy-and-paste, if the copied piece of code is mature and has few bugs, the resulting clone segments should likely have few bugs and would not experience many bug-fixing changes in the future. Therefore, we use history attributes to take into consideration the maturity of the copied piece of code. The maturity of a piece of code can be related to the time and the number of changes it experienced. Therefore, we consider six history attributes shown in Table 1.

Among the six history attributes in Table 1, the first three attributes aim to characterize the existing time and the changes of the copied piece of code itself, and the other three attributes aim to characterize the existing time and changes of the file containing the copied piece of code. Therefore, we consider both the maturity of the copied piece of code, as well as the maturity of its enclosing environment. We consider the attribute "recent changes" to differentiate the code pieces that experienced changes only in their early stage of existence with the code pieces which experienced changes recently, because the latter ones may be more immature (or they become unstable due to some changes to the whole software project recently).

3.3.2 Code Attributes

We use code attributes to consider the impact of the syntactical characteristics of the copied piece of code on whether the cloning operation requires consistency maintenance. Even if the copied piece of code is mature and has few bugs, it is still possible that the code will be changed due to the revision on the modules that the piece of code depends on. As demonstrated in the examples in Section 2, if the copied piece of code does not depend on many other parts in the code base, the clone group generated by the cloning operation

TABLE 1
History Attributes

Attribute	Definition
Existence Time (denoted as ET)	The period between the time of the appearance of the copied piece of code in the code base and the time of the cloning operation
Number of Changes	The number of changes that the copied piece of code has experienced in its evolution history
Number of Recent Changes	The number of changes that the copied piece of code has experienced recently. Currently, we deem $1/4$ of ET to be recent
File Existence Time (denoted as FET)	The period between the time of the appearance of the file containing the copied piece of code and the time of the cloning operation
Number of File Changes	The number of changes that the file containing the copied piece of code has experienced in the evolution history of the file
Number of Recent File Changes	The number of changes that the file containing the copied piece of code has experienced recently. Again, we currently deem $1/4$ of FET to be recent

will not be very likely to experience changes due to revisions in other parts of the code base. Therefore, for code attributes, we focus mainly on attributes related to code dependencies. Specifically, we consider eight code attributes as presented in Table 2.

The considered code attributes includes the lines of code, the number of field/parameter accesses, and the number of method invocations. We further consider different types of method invocations as attributes because some types of methods (e.g., library methods) may be much more stable than other methods. We use these attributes to characterize different ways the copied piece of code may depend on other parts. As we do not know how each type of dependency impacts the consistency maintenance beforehand, we consider all these attributes and rely on the construction algorithm of the Bayesian Network to weight these attributes.

3.3.3 Destination Attributes

We use destination attributes to consider the similarity between the context of the copied piece of code and the context of the pasting destination. Intuitively, if the context of the pasting destination is more similar to the context of the copied piece of code, the clone group generated by the operation may be more likely to experience consistent changes in the future. This is because the two clone segments may share more common dependencies and usages, so that changing these dependencies and usages may impact both clone segments. Therefore, we consider the seven destination attributes[6] in Table 3.

Generally, in our destination attributes, we measure the similarity between contexts by calculating the similarities between the names of enclosing files, the names of enclosing methods, etc. Again, when selecting destination attributes, we consider only whether each attribute might be related to the consistency maintenance of cloning operations. We rely on predictor construction to further sort out the relationships between the attributes.

3.4 Constructing the Predictor

The construction of our predictor includes three steps. First, we use a clone detector to identify a number of cloning operations performed in the version histories of existing software projects. Second, for each cloning operation acquired in the first step, we determine the values of the 21 attributes of the cloning operation and whether the cloning operation is consistency-maintenance-required or consistency-maintenance-free, thus forming a training instance. Third, we construct the Bayesian Network based on the training instances. The main process of the predictor construction phase is illustrated in Figure 1. In the figure, the code clone detector and the code tracker help us to fulfill the first step. The attribute extractor and the change tracker help us to fulfill the second step, while the model trainer helps us to fulfill the third step.

3.4.1 Collecting Existing Cloning Operations

The first step for constructing a predictor is to collect the cloning operations performed in the history of software projects. In this step, we first perform clone detection on each of the historical versions. Then, we build a code evolution genealogy based on the detected code clones. A clone evolution genealogy consists of a number of clone family trees, each of which represents the history of one clone group. Each node in a clone family tree corresponds to a clone group in a certain version of the project. In a clone family tree, the root node corresponds to a clone group that cannot be mapped to any clone groups in the previous version. If a clone group p in a version v_i can be mapped to another clone group p' in the previous version v_{i-1}, then the node corresponding to p is the child of the node corresponding to p' in a clone family tree. Kim et al. [8] proposed the concept of clone evolution genealogy as well as an approach to building a clone evolution genealogy by

6. It is possible that one cloning operation has multiple destinations. In such a case, for a boolean attribute, we set the value of the attribute as true if the attribute is true for at least one destination, and for a numeric attribute, we acquire the attribute for each destination, and use the maximal one as the attribute of the cloning operation.

TABLE 2
Code Attributes

Attribute	Definition
Number of Lines	The number of lines in the copied piece of code
Number of Invocations	The number of all method invocations in the copied piece of code
Number of Library Invocations	The number of library-method invocations in the copied piece of code
Number of Local Invocations	The number of invocations of methods defined in the same class with the copied piece of code
Number of Other Invocations	The number of invocations of methods that are neither from the library nor defined in the same class with the copied piece of code
Number of Field Accesses	The number of field accesses in the copied piece of code
Number of Parameter Accesses	The number of accesses to method parameters in the copied piece of code
Whether it is Test Code[a]	Whether the copied piece of code belongs to test code

a. In Microsoft, the code base of a software project typically contains a large proportion of test code. The dependence of test code on other code may typically be different from that of product code.

TABLE 3
Destination Attributes

Attribute	Definition
Whether it is a Local Clone	Whether the pasting destination and the copied piece of code are in the same file
File Name Similarity	The similarity between the name of the file containing the copied piece of code and the name of the file containing the pasting destination. Currently, we use the Levenshtein distance based similarity [30]. In all the following attributes, we also use Levenshtein distance to measure similarities between strings
Masked File Name Similarity	A variant of File Name Similarity. When the clone is local, the File Name Similarity has to be 1. But the meaning is quite different from where the File Name Similarity is close to 1. Therefore, we also use another attribute for file name similarity. For this attribute, the value is the same as the File Name Similarity when the cloning is not local, but 0 when the cloning is local
Method Name Similarity	The similarity between the name of the method containing the copied piece of code and the name of the method containing the pasting destination
Sum of Parameter Similarities (denoted as *SPS*)	Let us use *M1* and *M2* to denote the method containing the copied piece of code and the method containing the pasting destination, respectively. Supposing that *M1* has m parameters (whose names are denoted as P_1, P_2, ..., P_m) and *M2* has n parameters (whose names are denoted as Q_1, Q_2, ..., Q_n), we define *SPS* as $\sum_{i=1}^{m}\sum_{j=1}^{n} Sim(P_i, Q_j))$, where P_i denotes the name of the i-th parameter of *M1* and Q_j denotes the name of the the j-th parameter of *M2*
Maximal Parameter Similarity (denoted as *MPS*)[a]	We define *MPS* as *Max(Sim(*P_i*,* Q_j*))*, where $1<=i<=m$, $1<=j<=n$, and *Sim(x, y)* denotes the similarity between string x and string y
Difference in Only Postfix Numbers[b]	Whether the name of the method containing the copied piece of code and the name of the method containing the pasting destination differ in only their postfix numbers

a. We further consider the maximal parameter similarity because usually one pair of very similar parameters may be more informative than several pairs of moderately similar parameters.

b. We use this attribute because developers often use a list of methods with different postfix numbers in their names to indicate different versions of a method. This list of methods often contains many code clones but these clones seldom change consistently because only the method with the largest version number is going to change.

mapping code location (based on full source code file paths, and file comparison tools) of the code clones between each version and its previous version. We adopted the same approach in our implementation, and we used java-diff-utils (http://code.google.com/p/java-diff-utils/) for Java projects, and an internal file comparison tool of Microsoft for C# projects.

Based on the clone genealogy built with the above steps, we collect all the clone groups that correspond to the root nodes of clone family trees. Since these clone groups cannot be mapped to any clone groups in the previous version, we deem these clone groups as newly added by the developers through cloning operations. We refer to these clone groups as *original clone groups* in the rest of this paper. This means that, each original clone group corresponds to a cloning operation.

Another problem in recovering a cloning operation from an original clone group is that, we need to determine which clone segment in an original clone group is the copied piece of code and which clone segments corresponds to the duplicate code piece. We do the determination based on the following two heuristics:

- If a clone segment *seg* in an original clone group can be mapped to a code segment in the previous version, we determine that *seg* is a copied piece of code
- If none of the clone segments can be mapped to a code segment in the previous version, we

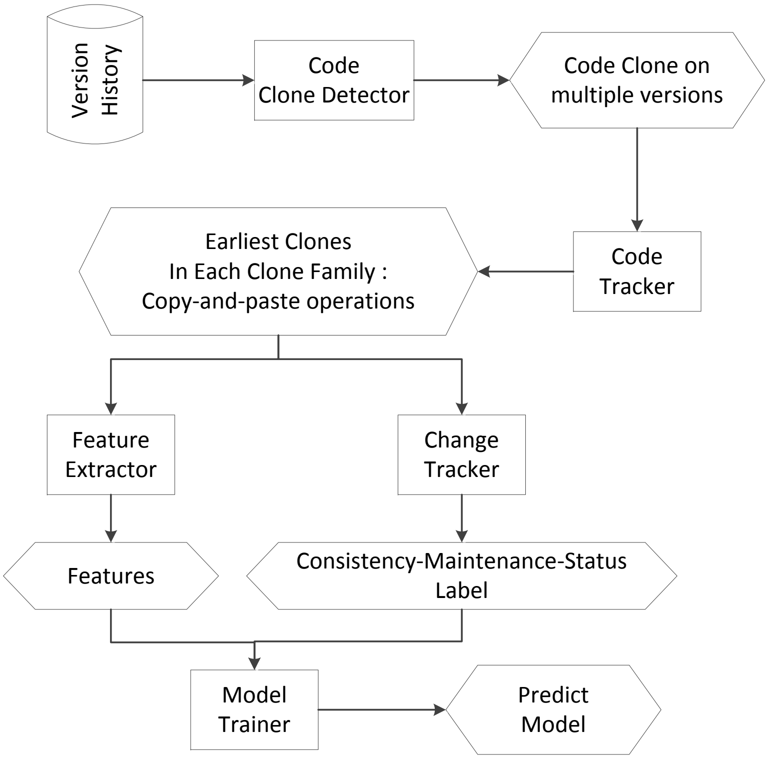

Fig. 1. The main process of the predictor construction phase

randomly choose a clone segment as the copied piece of code, because in this case, it is difficult to decide which segment is added first. Furthermore, choosing any segment as the copied piece has relatively little effect on our approach because these code segments will have similar attributes. First, these code segments have exactly the same history attributes. Second, these code segments have similar code attributes because they are code clones and share similar code structures. Third, they also have similar destination attributes because our destination attributes are symmetric (attributes will remain the same if we switch the original segment with the target segment). Thus, using any clone segment as the original segment will result in very similar attribute vectors. It should be noted that it is unlikely that more than one clone segment in an original clone group can be mapped to a code segment in the previous version. If so, the two mapped code segments in the previous version should be a clone group in the previous version. Thus, the existence of such a clone group is contradictory to our definition of original clone groups.

3.4.2 Determining Attribute Values and Consistency-Maintenance Status of Cloning Operations

To prepare the collected historical cloning operations as effective training instances of the prediction model, we also need to determine the values of the attributes and the consistency-maintenance status of each collected cloning operation. To determine the values of the history attributes, we analyze the version history of the copied piece of code in the cloning operation. In the version history analysis, we map code pieces in different versions in a similar way with which we map code clones in different versions. To determine the values of code attributes and destination attributes, we use a light-weight program analyzer to analyze the code version that the cloning operation is performed on, and extract the values of the attributes for the cloning operations. Specifically, our analyzer uses a parser to identify the enclosing methods of code clones, field/parameter accesses, invocations, and a def-use-chain based type inference tool to identify library and local invocations.

Since it is common for developers to make multiple changes between two successive versions and it is difficult to decide the order of the changes, we simply assume that any other changes will not impact the attributes of the cloning operations. As it is difficult to precisely measure the consistency-maintenance status of training instances, we classify the training instances into consistency-maintenance-required cloning operations and consistency-maintenance-free cloning operations for simplicity. That is to say, the consistency-maintenance status of a training instance is either 1 (for consistency-maintenance-required) or 0 (for consistency-maintenance-free). Note that, although our assumption for simplification may introduce some noise into our training data, the machine-learning technique we used can mitigate the impact of the noise in the training process. Specifically, we automatically determine the consistency-maintenance-status label of a training cloning operation based on the genealogy of the corresponding original clone group using the following two heuristics:

- If the clone group experiences no change or only divergent changes in the genealogy, we deem the corresponding cloning operation to be consistency-maintenance-free.
- If the clone group experiences at least one consistent change, we deem the corresponding cloning operation to be consistency-maintenance-required.

Based on the above heuristics, each code cloning operation is labeled as either consistency-maintenance-free or consistency-maintenance-required. The consideration behind our heuristics is that one consistent change can indicate extra consistency maintenance caused by code cloning operation. It is possible to apply our approach with other heuristics such as treating as consistency-maintenance-required only the cloning operations whose resulting clone groups experience consistent changes more than twice.

To determine whether an original clone group experience consistent changes in its genealogy, we adopt a procedure used in existing empirical studies on code clones [8]. For cloning operation *op*, our approach checks the nodes in the clone family tree *T*, whose root node corresponds to the original clone group *origin* generated by *op*. Obviously, except for the root node, each node *N* in *T* corresponds to a clone group *cg* that is evolved from *origin*, and *cg* can be mapped to another clone group cg_{-1} (which corresponds to *N*'s parent node in *T*) in the version prior to the version containing *cg*. Thus, we compare the clone segments in *cg* and cg_{-1} to see whether at least two segments in *cg* are changed from their corresponding segments in cg_{-1}. If so, we deem that a consistent change happens on cg_{-1}. Otherwise, we deem that there is no change or only an divergent change.

3.4.3 Training the Predictor

After we collect the cloning operations, and extract the attribute values and consistency-maintenance status of these operations, it is straightforward to feed all these data as input to the model training algorithms of Bayesian Networks (implemented by Weka), and generate a classification model as output. Then, we can use as classification model as the predictor of cloning operations for consistency-maintenance requirement.

3.5 Prediction

With the predictor constructed, we use it to predict whether an intended cloning operation requires consistency maintenance. When a developer intends to perform a cloning operation, our approach extracts the values of its attributes and uses the trained predictor to predict whether it requires consistency maintenance. Attribute extraction for an intended cloning operation is almost the same as attribute extraction for a performed cloning operation. The only difference is that, when extracting attributes for a performed cloning operation, we extract values for all its attributes at one time since both the copied piece of code and the paste destination are known; but for an intended cloning operation, we extract values of history attributes and code attributes when the developer copies a piece of code and extract values of the destination attribute when the developer pastes the copied code. The process of the predicting phase is depicted in Figure 2

Here, our Bayesian-Network-based predictor is able to provide a prediction score (between 0 and 1) depicting the probability that the intended cloning operation is consistency-maintenance-required. This score can thus help the developers to decide whether to perform the cloning operation. In practice, there can be an application-specific or domain-specific interpretation of the prediction scores to further help developers make the decision. For example, we can suggest that a developer treats cloning operations with a prediction score lower than a specific threshold as consistency-maintenance-free operations.

3.6 Usage Scenarios

Similar to most machine-learning based approaches, our approach may not provide a perfect prediction that exactly differentiates consistency-maintenance-required and consistency-maintenance-free code cloning operations. However, such an imperfect predictor can still provide useful advice to developers in their coding decisions. Specifically, we propose two scenarios where our approach may provide useful advice to developers in their decision-making of whether to perform a code cloning operation.

- *Conservative scenario*. In this scenario, developers are cautious and they tend to not perform any clones requiring consistency maintenance. So, the developers may consider using a relatively low threshold (e.g., 0.05) of prediction scores, so that a cloning operation is recommended only if its prediction score is even lower than the low

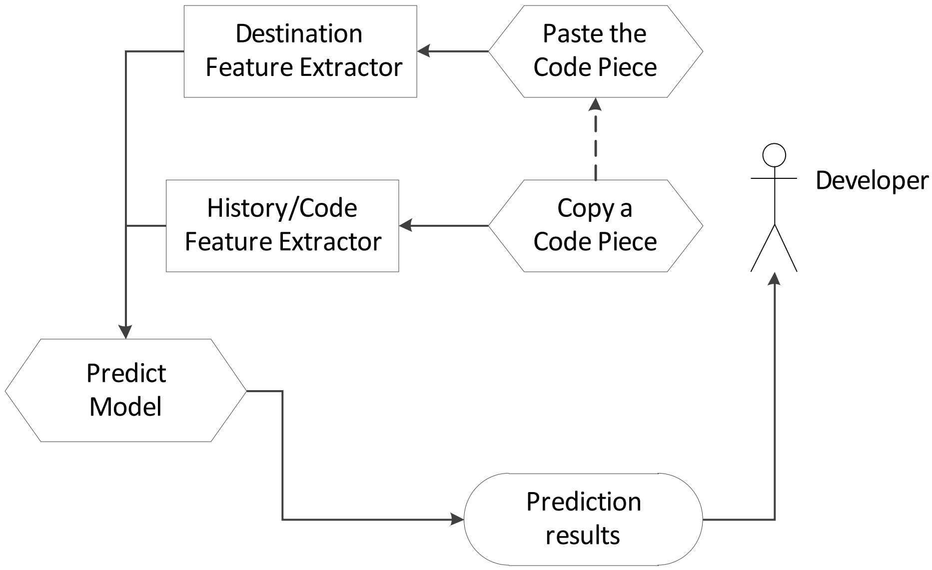

Fig. 2. The scenario of the predicting phase

threshold, which provide higher confidence when a cloning operation is predicted as consistency-maintenance-free.

- *Aggressive scenario*. In this scenario, developers may want to perform as many cloning operations as possible for quick development. Therefore, they are able to tolerate some extra maintenance effort, but they may also want to avoid as many consistency-maintenance-required operations as possible. So, the developers would choose a relatively high threshold (e.g., 0.5) of prediction scores, so that our approach will warn developers on a smaller number of cloning operations, which are more likely to be really consistency-maintenance-required.

It should be noted that our predictor provides suggestions to developers at the time of copy-and-paste. This is a proactive way for preventing consistency-maintenance-required cloning operations. Another possible time point of predicting whether a code cloning operation requires consistency maintenance is after the cloning operation is made but before the cloned code is checked into the version control system.

In this way, once a cloning operation is predicted as consistency-maintenance-required, the developer may either take actions immediately to remove it and then check in the revised code, or check in the code and conduct code refactoring at an appropriate time in future. In this way, we may even use the information of possible edits the developer makes after the code cloning operation and thus enhance our approach. In this paper, we do not use the information from further edits for prediction to make our approach more general and applicable to both ways.

4 Implementation

We implemented our approach as two prototypes. The first prototype is for Microsoft C# projects, and the second prototype is for open source Java projects. For simplicity, we refer to the first prototype as the Microsoft prototype, and the second prototype as the Open prototype.

In both prototypes, we automatically collect historical versions of some software projects. To save time and bandwidth consumption, we leverage an incremental strategy. Instead of downloading all historical versions one by one, we first download the initial version, and then continuously download the version differences to update it to the next version. During the process of the update, we backup all the versions updated to, so that we can get all the historical versions of a project. On these collected versions, we use off-the-shelf code-clone detectors to detect code clones, and construct the code clone genealogy and extract various attributes according to the approaches described in Section 3. Finally, we invoke Weka to train the prediction model.

The main differences between the two prototypes are as below.

1) The Microsoft prototype is written in C#, while the Open prototype is written in Java.
2) The Microsoft prototype interacts with the version control system of Microsoft to collect historical code versions, while the Open prototype currently supports collection of historical code versions from the SVN version control system. Note that it is easy to change the interface for other version control systems, such as git [7], and

7. http://git-scm.com/

CVS [8].

3) The Microsoft prototype uses the Microsoft parser and analyzer for C# source code to extract code attributes and destination attributes, while the Open prototype uses javaParser [9], an open source parser for Java source code, and we developed a light-weight program analyzer by ourselves to extract code attributes and destination attributes based on javaParser.
4) The Microsoft prototype uses a Microsoft tool (referred to as XIAO) [32] [12] for code clone detection in the historical versions of C# projects, while the Open prototype uses ConQAT [10], a scalable code-clone detector for multiple programming languages.

It should be noted that both XIAO and ConQAT are token-based code-clone-detection tools. Simply put, both tools transform the original program to a sequence of tokenized statements (i.e., different identifiers are converted to the same token in the statements), and apply hashing to statements so that identical statements are mapped to a same basket. The major difference of the two tools is as below. ConQAT applies hashing to a statement chunk (a sequence of tokenized statements) so that only identical statement chunks are considered to be code clones. By contrast, XIAO applies hashing to each tokenized statement, and generating code clones by combining neighboring identical statements based on a series of rough to fine strategies. Therefore, XIAO also considers near-miss clones which involve statement insertion, deletion, and reorder. To define the largest difference allowed between two clone segments, XIAO allows setting a similarity threshold, so that two clone segments are considered to be a code clone only if their similarity is larger than the similarity threshold. For both prototypes, we use the default settings of the code-clone-detection tools. Specifically, for XIAO, we are using 20 lines as the minimal clone size, and 0.9 as the similarity threshold; while for ConQAT, we are using 10 lines as the minimal clone size. Since ConQAT uses a stricter criterion for code clones, it is reasonable for ConQAT to have a smaller minimal clone size by default.

5 EVALUATION

To investigate the effectiveness of our approach, we carried out our evaluation on four projects, including two large industrial software projects from Microsoft (denoted as XProj[11] and YProj in this paper), and two open source software projects: JFreechart and tuxGuitar[12]. To make our evaluation subjects more representative, we chose projects from both industry and the open source community. Furthermore, these four projects are written in two different programming languages (i.e., projects from Microsoft are written in C#, while the open source projects are written in Java), from different domains, and with different code sizes.

In this section, we first introduce the methodology for our evaluation in Section 5.1. Then, we present the evaluation of our approach on the two Microsoft projects in Section 5.2, and the evaluation of our approach on the two open source projects in Section 5.3. We summarize our evaluation results in Section 5.5. We discuss the threats to validity in Section 5.6.

5.1 Methodology

We evaluate the effectiveness of our approach from the following four perspectives.

Effectiveness in the conservative scenario. In the conservative scenario, we set a threshold (close to 0) and all the cloning operations with a prediction value lower than the threshold are deemed consistency-maintenance-free cloning operations. Then we measure the effectiveness of our approach using the following two metrics:

- Recommendation rate: the proportion of cloning operations that are predicted as consistency-maintenance-free in all cloning operations for prediction.
- Precision: the proportion of true consistency-maintenance-free operations in the cloning operations that are predicted by our approach as consistency-maintenance-free.

In this scenario, developers do not want to concede consistency-maintenance-required cloning operations while they can tolerate some potential consistency-maintenance-free cloning operations being not performed, so the precision is expected to be near 100% and the recommendation rate is not expected to be so high.

Effectiveness in the aggressive scenario. In the aggressive scenario, we set a certain threshold (close to 1) and all the cloning operations with a predicted value higher than the threshold are deemed as consistency-maintenance-required cloning operations. Then, we measure the effectiveness of our approach using the following two metrics:

- Warning rate: the proportion of cloning operations that are predicted as consistency-maintenance-requried in all cloning operations for prediction.
- Recall: the proportion of consistency-maintenance-required operations that are

8. http://sourceforge.net/apps/trac/sourceforge/wiki/CVS
9. http://code.google.com/p/javaparser/
10. https://www.conqat.org/
11. According to Microsoft regulations, we are unable to disclose the names and the application domains of the two projects.

12. Experimental data for open source projects is available at https://sourceforge.net/projects/transtrl/files/ClonePrediction.

predicted to be consistency-maintenance-required among all actually consistency-maintenance-required operations in the cloning operations for prediction.

In this scenario, developers tend to perform as many cloning operations as possible, so they will only care about the cloning operations that are predicted as consistency-maintenance-required with high confidence. Therefore, it is expected that the recall value should be significantly higher than the value of the warning rate.

Contributions of the three types of attributes. We studied the impact of different attribute groups on the effectiveness of our approach. This provides insight into how the underlying attributes contribute to the predictor. We use the same metrics defined above to understand the contributions of the three groups of attributes.

Feasibility for cross-project prediction. Moreover, developers may want to apply our approach on a new project that has too short a version history for our approach to collect training data. In such a case, one possible alternative solution would be cross-project prediction, in which the historical cloning operations of one project are used to predict the consistency maintenance of the cloning operations in another project. Therefore, we further investigate whether our approach can help when performing cross-project prediction.

In our evaluation, we are using different metrics for different application scenarios, because software developers may care about different aspects of the prediction results. Specifically, in conservative scenarios, developers want to guarantee a high precision and maximize the recommendation rate, while in aggressive scenarios, developers want to keep a low warning rate, and maximize the recall. However, for the ease of understanding, as a complement, we provide both precision and recall information in all the experimental results.

5.2 Evaluation on Microsoft Projects

We present the evaluation of our approach on two Microsoft projects as below. The information about the two Microsoft projects are shown in Table 4. The second column of Table 4 presents the start date of each project. The third column presents the date of the last version of each project used for our evaluation. The fourth column presents the size range of each project (in KLOC, i.e., kilo lines of code) between the start date and the end date in columns 2 and 3. It should be noted that, in the whole evaluation section, the "start date" of a project refers to the date when the current code repository is initialized or when the legacy code is imported to the current code repository, while the "end date" of a project refers to the date of the last version that is available when we performed our evaluation on the project.

TABLE 4
Microsoft Projects Used in Our Evaluation

Project	Start Date	End Date	KLOC
XProj	Oct-31-2005	Dec-27-2010	0 to 4,521
YProj	Oct-01-2007	Dec-27-2010	987 to 1,073

For each software project, we downloaded all the versions (i.e., weekly snapshots) between the start date and the end date of the software project. We used weekly snapshots because the number of code submissions for the projects are very large and it is difficult to process them one by one in reasonable time. For each project, we collected cloning operations from only the older versions (which had existed in the software projects for more than two years at the time of collection) because there may not be enough time for us to observe consistent or divergent changes in newly generated clone groups. It should be noted that, for the software projects that involves legacy code (i.e., YProj), we did not collect cloning operations from the code clones that are in the initial version (the code version on the "start date") of the software project. The reason is that, these code clones may be generated long time before the earliest version available to us, so that we are unable to estimate how the cloning operations are performed, and whether they experience consistent changes.

After that, we extracted the consistency-requirement-status label and values of the attributes of all these collected cloning operations (as described in Section 3.4) to build a data set, which provides both the training data and the testing data. Finally, we performed 10-fold cross validation [20] on the data set to acquire the effectiveness of our approach. Table 5 shows the detail about the data sets that we extracted from the four software projects. In Table 5, columns 1-4 present the project name, the number of collected cloning operations, the number of consistency-maintenance-required cloning operations, and the number of consistency-maintenance-free cloning operations. The numbers in the bracket in columns 3-4 present the proportion of consistency-maintenance-required and consistency-maintenance-free operations in all cloning operations, respectively.

From Table 5, we can observe that, in both software projects, the number of consistency-maintenance-free cloning operations is much greater than the number of consistency-maintenance-required cloning operations. This observation is consistent with the findings of the empirical study by Göde and Koschke [21].

5.2.1 Effectiveness in the Two Scenarios

Table 6 depicts the effectiveness of our classification on consistency-maintenance-required /consistency-maintenance-free cloning operations without considering usage scenarios. As 0.5 is the default threshold

TABLE 5
Extracted Data Sets from Microsoft Projects

Project	#Cloning Operations	#Consistency-Maintenance-Required (%)	#Consistency-Maintenance-Free (%)
XProj	3407	502(14.7%)	2905(85.3%)
YProj	401	76(19.0%)	325(81.0%)

TABLE 6
Average Classification Effectiveness on Microsoft Projects

Project (Threshold)	Avg. Precision	Avg. Recall
XProj (0.5)	68.3%	51.6%
YProj (0.5)	62.9%	70.0%

of all classifiers, we also use 0.5 as the threshold. The columns 2 and 3 in Table 6 show the average precision and recall of two categories of cloning operations, respectively. Specifically, for average precision, we add the precision for consistency-maintenance-required cloning operations with that for consistency-maintenance-free cloning operations, and divide the sum by 2. For the average recall, we do the same thing. From the table, we can see that our approach yields reasonable precision and recall for both projects, but it is hard to know how our approach is able to help developers. Therefore, we further evaluate our approach on the two usage scenarios.

Table 7 depicts the effectiveness of our approach for the *conservative scenario* using the following threshold values: 0.2, 0.15, 0.1, 0.05, and 0.01. In Table 7, column 1 presents the combination of the project and the threshold value, column 2 presents the recommendation rate, and column 3 presents the precision of our approach. In column 4, we also present the recall of our approach.

From Table 7, we can make the following observation. With the threshold value of 0.05, our approach is able to predict about 50% to 60% of the cloning operations as consistency-maintenance-free with a precision higher than 94.9%. This demonstrates the value of our approach in the conservative scenario: Our approach provides a quite accurate suggestion of which cloning operations are consistency-maintenance-free for more than 50% of the cloning operations. Without our approach, it is difficult for developers to make such decisions confidently. Therefore, our approach provides conservative developers some guidance for identifying consistency-maintenance-free cloning operations with high precision. Additionally, the recall of our approach is around 60%, which shows that we can recommend a large portion of consistency-maintenance-free cloning operations as consistency-maintenance-free.

Table 8 shows the effectiveness of our approach for the *aggressive scenario* using the following threshold values: 0.9, 0.8, 0.7, 0.6, and 0.5. In Table 8, column 1 presents the combination of the project and the threshold value, column 2 presents the warning rate, and column 3 presents the recall. In column 4, we present the precision of our approach.

From Table 8, we can make the following two observations. First, in both projects, under most of the thresholds, the warning rate of our approach is less than half of the corresponding recall. Using 0.5 as the threshold, in XProj, our approach recommends 19.7% of cloning operations as consistency-maintenance-required to cover 59% real consistency-maintenance-required cloning operations, while in YProj, our approach recommends 39.9% of cloning operations as consistency-maintenance-required to cover 72.4% real consistency-maintenance-required cloning operations. It should be noted that, without our approach, the warning rate and the recall should be about the same, as the data used in our evaluation already reflects the practice of deve2lopers to perform cloning operations. Therefore, this observation demonstrates the value of our approach in the aggressive scenario: With the guidance of our approach, developers may avoid a significant percentage of consistency-maintenance-required cloning operations by not doing a small percentage of cloning operations. Also, the precision of our approach is between 30% and 60%, which is acceptable because the data in our problem is rather imbalanced[13]. Second, the experimental results for the two projects have different warning rate for the same threshold. This indicates that a different threshold may be used for different projects to better utilize our approach.

5.2.2 Impacts of Attribute Groups

As our approach uses three groups of attributes to predict whether intended cloning operations require consistency maintenance, we removed each group of attributes at one time to check how each group of attributes contributes to the overall effectiveness of our approach. We experimented with three variants of our approach, each using two groups of attributes. Again, we considered two scenarios: the *conservative scenario* and the *aggressive scenario*.

With a fixed threshold, it may be difficult to compare the different variants of our approach. For example, when under a certain threshold, variant A

13. Imbalanced data has been a well-known difficult problem in machine learning [17] [18]. Consistency-maintenance-required cloning operations only account for 10% to 20% of all cloning operations, so that random classification only gets 10% to 20% precision & recall for consistency-maintenance-required cloning operations.

TABLE 7
Effectiveness in the *Conservative Scenario* on Microsoft Projects

Project (Threshold)	Recommendation Rate	Precision	Recall
XProj(0.01)	41.3%	96.4%	46.7%
XProj(0.05)	60.1%	94.9%	66.9%
XProj(0.10)	67.0%	93.9%	73.8%
XProj(0.15)	68.7%	93.7%	75.5%
XProj(0.20)	71.0%	93.2%	77.6%
YProj(0.01)	46.6%	96.3%	55.4%
YProj(0.05)	51.9%	95.7%	61.3%
YProj(0.10)	54.4%	94.0%	63.1%
YProj(0.15)	55.4%	94.1%	64.3%
YProj(0.20)	56.4%	93.4%	65.0%

TABLE 8
Effectiveness in the *Aggressive Scenario* on Microsoft Projects

Project (Threshold)	Warning Rate	Recall	Precision
XProj(0.9)	7.3%	33.9%	68.4%
XProj(0.8)	11.9%	45.0%	55.7%
XProj(0.7)	14.5%	48.6%	49.4%
XProj(0.6)	18.1%	57.0%	46.4%
XProj(0.5)	19.7%	59.0%	44.1%
YProj(0.9)	23.2%	55.3%	45.2%
YProj(0.8)	30.7%	64.5%	39.8%
YProj(0.7)	34.4%	67.1%	36.7%
YProj(0.6)	38.2%	69.7%	34.6%
YProj(0.5)	39.9%	72.4%	34.4%

TABLE 10
Recall of the Three Variants in the *Aggressive Scenario* on Microsoft Projects

Project (Warning Rate)	All	Without History	Without Code	Without Destination
XProj(10%)	41.2%	40.4%	28.7%	33.3%
XProj(20%)	59.4%	57.6%	54.8%	51.2%
XProj(30%)	68.5%	68.5%	62.7%	62.5%
YProj(10%)	32.9%	34.2%	15.8%	26.3%
YProj(20%)	53.9%	53.9%	35.5%	39.5%
YProj(30%)	63.2%	63.2%	61.8%	53.9%

can avoid 10% of consistency-maintenance-required cloning operations by warning developers to be cautious on 5% of cloning operations, while variant B can avoid 20% of consistency-maintenance-required cloning operations by recommending developers to not perform 10% of cloning operations. It would be difficult to judge which variant is better. Therefore, in the comparison, we fixed the recommendation rate as 50%, 60%, and 70% (i.e., in the recommendation rate range of our approach using thresholds between 0.2 and 0.01) for the *conservative scenario* and fixed the warning rate as 10%[14], 20%, and 30% (i.e., in the warning rate range of our approach using thresholds between 0.9 and 0.5) for the *aggressive scenario*. Tables 9 and 10 show the results of comparing the three variants with our approach (i.e., using all three groups of attributes) for the two Microsoft projects.

14. Note that a 10% warning rate is equivalent to a 90% recommendation.

From Tables 9 and 10, we can make the following observations. First, in most cases, removing the history attributes has little impact on the prediction in both scenarios. Second, removing the code attributes results in a small impact in the *conservative scenario* and some negative impact in the *aggressive scenario*. Third, for both scenarios, removing the destination attributes results in some negative impacts on the effectiveness. Fourth, removing history attributes may even slightly improve the results in some scenarios due to possible noise in those attributes.

5.2.3 Effectiveness of Cross-Project Prediction

In Section 5.2.1, our cross-validation is based on each project individually. For each subject project, we divided the collected cloning operations into the training set and the testing set. Then we used the training set to train a predictor and tested it on the testing set. However, if developers would like to leverage our approach at the beginning of their project, there will not be enough training data from the project. In such a case, the developers may have to use a predictor trained from the data of another software project, which we refer to as cross-project prediction. Cross-project prediction is notoriously difficult for many mining-based software engineering approaches [1], since software projects are often very different from each other in their usages, structures, programming rules, etc. What makes the situation even worse is that it is difficult to measure and handle such differences. Therefore, it would be interesting to study whether

TABLE 9
Precision of the Three Variants in the *Conservative Scenario* on Microsoft Projects

Project (Recommendation Rate)	All	Without History	Without Code	Without Destination
XProj(50%)	95.8%	95.6%	95.9%	92.7%
XProj(60%)	94.9%	94.5%	94.8%	92.5%
XProj(70%)	93.4%	93.4%	92.2%	92.1%
YProj(50%)	96.5%	95.0%	95.5%	93.5%
YProj(60%)	91.3%	91.3%	91.3%	91.3%
YProj(70%)	90.0%	90.0%	89.6%	87.5%

our approach still provides help for cross-project prediction.

To study the effectiveness of our approach on cross-project prediction, we evaluated our approach by training our predictor with the data from one of the Microsoft project and tested the predictor on the other project. We present the evaluation results for both the conservative scenario and the aggressive scenario in Tables 11 and 12, respectively. In both tables, we use label "A-B" to denote training on project "A" and testing on project "B".

From Table 11, we can observe that, with a threshold of 0.05, for the setting of XProj-YProj our approach recommends 60% cloning operations as consistency-maintenance-free with a precision of 91.5%, and for the setting of YProj-XProj, our approach recommends 33.1% cloning operations as consistency-maintenance-free with a precision of 94.1%. Compared with Table 7, we observe that the effectiveness of our approach on cross-prediction drops. However, in YProj, the proportion of consistency-maintenance-free cloning operations is 81%, and in XPoj, the proportion is 85.5%. Therefore, our approach can still enhance the precision by 10.5% for YProj, and 8.6% for XProj (note that the improvement is not trivial considering that the precision is already very high). Furthermore, we observe that the effectiveness of our approach can be improved if we can appropriately tune the threshold. For example, with a threshold of 0.01, our approach with the XProj-YProj setting recommends more than 50% of cloning operations as consistency-maintenance-free with a precision of 93.8%.

Similarly, comparing Table 12 and Table 8, we can observe that, our approach is also significantly less effective in the *aggressive scenario* for cross-project prediction than for inner-project prediction. With a threshold of 0.5, for the setting of XProj-YProj, our approach warns developers on 16.5% of cloning operations, and may help to avoid 32.9% of consistency-maintenance-required cloning operations, For the setting of YProj-XProj our approach warns developers on 45.7% of cloning operations, and may help to avoid 77.9% of consistency-maintenance-required cloning operations. For the setting of YProj-XProj, the recall of our approach still almost doubles the corresponding warning rate.

TABLE 13
Open Source Projects Used in Our Evaluation

Project	Start Date	End Date	KLOC
jFreeChart	Nov-17-2002	Aug-26-2012	0 to 328
tuxGuitar	Apr-17-2008	Aug-26-2012	0 to 146

5.3 Evaluation on Open Source Projects

Besides Microsoft projects, we further evaluate our approach on two open source projects: jFreeChart and tuxGuitar. jFreeChart is a popular java library for drawing various charts, while tuxGuitar is a popular tablature editor. The basic information of the two projects is shown in Table 13. As in Table 4, the columns in the table presents the name, start date, end date and size of the projects.

Similar to our experiments on Microsoft projects, for each software project, we downloaded all weekly snapshots between the start date and the end date, and collected cloning operations from only the versions that had existed in the software projects for more than two years at the time of collection. Table 14 shows the detail about the data sets that we extracted from the two software projects. From Table 14, we confirm that the number of consistency-maintenance-free cloning operations is much greater than the number of consistency-maintenance-required cloning operations, which is consistent with our observation on Microsoft projects, and the findings of the empirical study by Göde and Koschke [21].

5.3.1 *Effectiveness in the Two Scenarios*

Table 15 depicts the effectiveness of our classification on consistency-maintenance-required /consistency-maintenance-free cloning operations without considering usage scenarios. Similar to our evaluation on Microsoft projects, the columns 2 and 3 in Table 15 show the average precision and recall of two categories of cloning operations, respectively. From the table, we can also see that our approach yields reasonable precision and recall for both projects, and we need to further evaluate our approach on the two usage scenarios to understand how much our approach is able to help developers.

The effectiveness of our approach on the two scenarios (i.e., the Conservative Scenario and the Aggressive Scenario) with different thresholds, as depicted

TABLE 11
Effectiveness for Cross-Project Prediction in the *Conservative Scenario* on Microsoft Projects

Setting(Threshold)	Recommendation Rate	Precision	Recall
XProj-YProj(0.01)	52.4%	93.8%	60.6%
XProj-YProj(0.05)	61.6%	91.5%	69.5%
XProj-YProj(0.10)	66.3%	90.2%	73.8%
XProj-YProj(0.15)	67.8%	89.3%	74.7%
XProj-YProj(0.20)	70.3%	88.7%	76.9%
YProj-XProj(0.01)	22.5%	95.3%	25.1%
YProj-XProj(0.05)	33.1%	94.1%	36.5%
YProj-XProj(0.10)	40.8%	94.4%	45.2%
YProj-XProj(0.15)	41.2%	94.4%	45.6%
YProj-XProj(0.20)	43.6%	94.3%	48.2%

TABLE 12
Effectiveness for Cross-Project Prediction in the *Aggressive Scenario* on Microsoft Projects

Setting(Threshold)	Warning Rate	Recall	Precision
XProj-YProj(0.9)	0.5%	1.3%	49.3%
XProj-YProj(0.8)	3.0%	3.9%	24.6%
XProj-YProj(0.7)	4.0%	5.3%	25.1%
XProj-YProj(0.6)	5.5%	7.9%	27.2%
XProj-YProj(0.5)	16.5%	32.9%	37.8%
YProj-XProj(0.9)	22.9%	52.3%	33.7%
YProj-XProj(0.8)	33.3%	67.3%	29.8%
YProj-XProj(0.7)	34.0%	68.1%	29.5%
YProj-XProj(0.6)	42.3%	72.1%	25.1%
YProj-XProj(0.5)	45.7%	77.9%	25.1%

TABLE 14
Extracted Data Sets from Open Source Projects

Project	#Cloning Operations	#Consistency-Maintenance-Required (%)	#Consistency-Maintenance-Free (%)
JFreeChart	1148	89(7.8%)	1059(92.2%)
tuxGuitar	444	60(13.5%)	384(86.5%)

TABLE 15
Average Classification Effectiveness on Open Source Projects

Project (Threshold)	Avg. Precision	Avg. Recall
JFreeChart (0.5)	58.3%	63.0%
tuxGuitar (0.5)	61.5%	60.1%

in Table 16, and Table 17. In the tables, we provide both precision and recall (in different order because in either scenario, developers care more about one of them) information.

From Table 16, we can observe that, in both projects, with the threshold value of 0.05, our approach is able to predict about 50% to 60% of the cloning operations as consistency-maintenance-free with a precision higher than 94.0%. Also, the recall of our approach is between 50% and 60%, which shows that our approach correctly identifies a large portion of consistency-maintenance-free cloning operations.

From Table 17, we can observe that, 1) similar to our evaluation results on Microsoft projects, the warning rate of our approach is less than half of the corresponding recall in most cases; and 2) at the threshold of 0.5, our approach warns developers on 13% to 18% cloning operations, which may help to avoid 37% to 45% of real consistency-maintenance-required cloning operations. Therefore, our evaluation on open source projects confirms the effectiveness of our approach on both scenarios. The precisions of our approach for both projects are between 20% and 30%, which is relatively low. However, the experimental data (training and testing sets) is even more imbalanced in open source software projects (about 10% of cloning operations are consistency-maintenance-required), so it is more difficult to achieve high precision for the minor category (i.e., a random classification will have only a precision & recall of 10% for consistency-maintenance-required cloning operations). Table 15 shows that our average precision and recall for two categories of cloning operations are still acceptable. Furthermore, in the aggressive scenario, precision is not as important as warning rate and recall.

It should be noted that, for the aggressive scenario, a same threshold may result in different warning rate in different projects. Therefore, although a fixed threshold such as 0.5 is able to provide help to developers, some tuning of the threshold to maintain a similar warning rate in different projects will also benefit developers. We will discuss the potential approaches to tune the threshold in Section 6.5.

TABLE 16
Effectiveness in the *Conservative Scenario* on Open Source Projects

Project (Threshold)	Recommendation Rate	Precision	Recall
JFreeChart(0.01)	35.1%	97.9%	37.3%
JFreeChart(0.05)	53.0%	95.9%	55.1%
JFreeChart(0.10)	62.1%	95.6%	64.4%
JFreeChart(0.15)	66.1%	95.5%	68.4%
JFreeChart(0.20)	70.7%	94.9%	72.7%
tuxGuitar(0.01)	40.5%	94.4%	44.2%
tuxGuitar(0.05)	56.3%	94.0%	61.2%
tuxGuitar(0.10)	65.5%	92.1%	69.8%
tuxGuitar(0.15)	68.7%	92.1%	73.2%
tuxGuitar(0.20)	70.0%	92.3%	74.7%

TABLE 17
Effectiveness in the *Aggressive Scenario* on Open Source Projects

Project (Threshold)	Warning Rate	Recall	Precision
JFreeChart(0.9)	1.7%	7.0%	31.9%
JFreeChart(0.8)	4.0%	15.6%	30.2%
JFreeChart(0.7)	6.1%	20.2%	25.6%
JFreeChart(0.6)	9.8%	28.7%	22.7%
JFreeChart(0.5)	13.1%	37.0%	21.9%
tuxGuitar(0.9)	5.4%	6.7%	16.8%
tuxGuitar(0.8)	8.8%	20.0%	30.7%
tuxGuitar(0.7)	12.4%	28.3%	30.8%
tuxGuitar(0.6)	17.8%	43.3%	32.9%
tuxGuitar(0.5)	18.9%	45.0%	32.2%

TABLE 19
Recall of the Three Variants in the *Aggressive Scenario* on Open Source Projects

Project (Warning Rate)	All	Without History	Without Code	Without Destination
JFreeChart(10%)	29.4%	32.6%	22.5%	33.7%
JFreeChart(20%)	49.5%	57.3%	43.8%	48.3%
JFreeChart(30%)	67.0%	71.9%	62.9%	66.3%
tuxGuitar(10%)	20.0%	20.0%	10.0%	20.0%
tuxGuitar(20%)	46.7%	46.7%	25.0%	48.3%
tuxGuitar(30%)	60.0%	60.0%	43.3%	61.7%

5.3.2 Impacts of Attribute Groups

For the two open source projects, we also studied the impact of different attribute groups, and presented the results in Table 18, and Table 19. From Tables 18 and 19, we can make the following observations.

First, removing the history attributes has small impacts on the prediction for the project tuxGuitar, which is consistent with our observation on Microsoft projects. For the subject JFreeChart, removing history attributes results in non-trivial improvement in both scenarios. These observations indicate that it may be feasible to use only the code attributes and the destination attributes to predict whether intended cloning operations require consistency maintenance, or to develop more flexible attribute sets for different software projects. Second, removing the code attributes results in a small impact in the *conservative scenario* but a significantly negative impact in the *aggressive scenario*, which is also consistent with our findings on Microsoft projects. Third, removing the destination attributes has small impacts in the two open source projects. By contrast, it results in significantly negative impacts on the effectiveness in Microsoft projects.

5.3.3 Effectiveness of Cross-Project Prediction

In this subsection, we present the result of cross-project prediction between the two open source projects. We did cross-project prediction separately for open source projects and Microsoft projects, because there are large difference between Microsoft projects and open source projects (in software domains, programming styles, programming language used, and clone-detection tool used).

Similarly, we use label "*A-B*" to denote training on project "*A*" and testing on project "*B*", and we present the evaluation results for both the conservative scenario and the aggressive scenario in Tables 20 and 21, respectively.

From Table 20, we can observe that, for the setting of JFreeChart-tuxGuitar, our approach recommends developers to perform 67% cloning operations with a precision of 91.0%, and for the setting of tuxGuitar-JFreeChart, our approach recommends developers to perform 50% cloning operations with a precision 94.1%. Compared with Table 16, we also observe that the effectiveness of our approach on cross-prediction drops. The proportion of consistency-maintenance-free cloning operations is 86.5% for tuxGuitar, and 92.4% for JFreeChart. Therefore, the enhancement of our approach over the developers' initial decision is 4.5% and 1.7%, respectively. Compared to the enhancements observed in the cross-project prediction

TABLE 18
Precision of the Three Variants in the *Conservative Scenario* on Open Source Projects

Project (Recommendation Rate)	All	Without History	Without Code	Without Destination
JFreeChart(50%)	96.5%	97.4%	97.2%	96.3%
JFreeChart(60%)	95.5%	97.4%	96.2%	96.7%
JFreeChart(70%)	95.0%	96.9%	95.9%	96.3%
tuxGuitar(50%)	93.7%	93.7%	93.2%	93.7%
tuxGuitar(60%)	92.1%	92.1%	92.5%	92.9%
tuxGuitar(70%)	92.3%	92.3%	89.4%	92.6%

TABLE 20
Effectiveness for Cross-Project Prediction in the *Conservative Scenario* on Open Source Projects

Setting(Threshold)	Recommendation Rate	Precision	Recall
JFreeChart-tuxGuitar(0.01)	15.8%	97.1%	17.7%
JFreeChart-tuxGuitar(0.05)	67.3%	91.0%	70.8%
JFreeChart-tuxGuitar(0.10)	76.1%	91.1%	80.2%
JFreeChart-tuxGuitar(0.15)	78.0%	90.8%	81.9%
JFreeChart-tuxGuitar(0.20)	81.8%	90.4%	85.5%
tuxGuitar-JFreeChart(0.01)	30.9%	95.8%	32.1%
tuxGuitar-JFreeChart(0.05)	50.0%	94.1%	51.0%
tuxGuitar-JFreeChart(0.10)	58.7%	93.2%	59.3%
tuxGuitar-JFreeChart(0.15)	73.3%	93.9%	74.6%
tuxGuitar-JFreeChart(0.20)	74.5%	93.7%	75.7%

between the two Microsoft projects, our approach performs worse in the cross-prediction between open source projects. This may due to the less similarity between open source projects, compared to the similarity between two Microsoft projects.

Similarly, comparing Table 21 and Table 17, we can observe that, for open source projects, our approach is also significantly less effective in the *aggressive scenario* for cross-project prediction than for inner-project prediction. For the setting of JFreeChart-tuxGuitar, our approach warns developers on 9.0% of cloning operations and may help to avoid 26.7% of consistency-maintenance-required cloning operations, and for the setting of tuxGuitar-JFreeChart, our approach warns developers on 8.9% of cloning operations and may help to avoid 12.4% of consistency-maintenance-required cloning operations. The precision for JFreeChart-tuxGuitar is acceptable, but the precision for tuxGuitar-JFreeChart is very low. Such results indicate that we may still need further improvements to our approach (e.g., selection of proper training software projects, tuning of thresholds) to make it practically useful for cross-project prediction in open source software projects. We will discuss this issue in Section 6.

5.4 Examples of Prediction Results

To provide better understanding of how our approach is able to help developers, in this subsection, we present two concrete examples of our prediction results on the experimented subjects.

Example 1. This example is from XYPlot.java in the JFreeChart project, version No. 1295. In this example, the developers copied the code piece below, and pasted it in the same file with no change. After the cloning operation, the two clone segments experienced three consistent changes: 1) changing `itemBounds[0]` on Line 5 to `itemBounds[0] + 1`, 2) adding an argument to the method `drawItem(...)` on Line 11, and 3) adding an "If" condition before Line 11 to make the invocation of `drawItem(...)` conditionally. For this cloning operation, our approach extracts all three categories of attributes as follows. 1) The copied code piece has 48 historical versions, and 1 recent change; 2) the code piece includes 4 parameter accesses, 7 method invocations and none of the invocations are of library methods; and 3) it is a local clone. Finally, our approach gives the cloning operation a score 0.651, which is a relatively high score so that our approach will warn developers to be cautious on performing this cloning operation.

```
1  if(state.getProcessVisibleItemsOnly()){
2    int[] itemBounds = RendererUtilities.findLiveItems(
3        dataset, series, xAxis.getLowerBound(),
4        xAxis.getUpperBound());
5    firstItem = itemBounds[0];
6    lastItem = itemBounds[1];
7  }
8  state.startSeriesPass(dataset, series, firstItem,
9      lastItem, pass, passCount);
10 for(int item = firstItem; item <= lastItem; item++){
11   renderer.drawItem(g2, state, dataArea, info,
12       this, xAxis, yAxis, dataset, series, item,
13       crosshairState, pass);
14 }
15 state.endSeriesPass(dataset, series, firstItem,
16     lastItem, pass, passCount);
```

Example 2. This example is from TGBrowserConnection.java in the project tuxGuitar, version 374. In this example, the developers copied the code piece below, and pasted it in the same file with slight change. After the cloning operation, the two clone segments experienced no changes. For this cloning

TABLE 21
Effectiveness for Cross-Project Prediction in the *Aggressive Scenario* on Open Source Projects

Setting(Threshold)	Warning Rate	Recall	Precision
JFreeChart-tuxGuitar(0.9)	0.0%	0.0%	0.0%
JFreeChart-tuxGuitar(0.8)	2.0%	6.7%	45.3%
JFreeChart-tuxGuitar(0.7)	2.0%	6.7%	45.3%
JFreeChart-tuxGuitar(0.6)	2.9%	6.7%	31.2%
JFreeChart-tuxGuitar(0.5)	9.0%	26.7%	40.1%
tuxGuitar-JFreeChart(0.9)	3.0%	4.5%	11.6%
tuxGuitar-JFreeChart(0.8)	5.0%	4.5%	7.0%
tuxGuitar-JFreeChart(0.7)	7.8%	9.0%	8.9%
tuxGuitar-JFreeChart(0.6)	7.8%	9.0%	8.9%
tuxGuitar-JFreeChart(0.5)	8.9%	12.4%	10.8%

operation, our approach extracts all three categories of attributes as follows. 1) The copied code piece had 143 historical versions, and no changes; 2) the code piece includes 10 method invocations and 4 of them are of library methods; and 3) it is also a local clone. Finally, our approach give the cloning operation a score 0.015, which is a very low score and will be passed for most thresholds.

```
new Thread(new Runnable(){
  public void run(){
    try{
      if(isOpen()){
        getBrowser().cdUp();
        notifyCd(callId);
      }else{
       notifyClosed(callId);
      }
    }catch (TGBrowserException e){
      notifyError(callId,e);
      e.printStackTrace();
    }
  }
}).start();
```

From the above examples, we can see that, the attributes of the two examples are different, but it is not easy to tell explicit rules that indicates consistency maintenance is required or not. In fact, it may not be easy for the developers also to manually predict whether intended code clones require consistency maintenance. In such cases, our approach can not only provide useful information (the collected attributes), but also present a meaningful reduction for the potential of the intended code cloning operations.

5.5 Summary

To sum up, based on the evaluation of our approach on two Microsoft projects and two open source projects, we have the following major findings.

- Our approach to predicting whether a cloning operation requires consistency maintenance is effective in general. In the conservative scenario, with our approach, developers may take advantage of more than half of cloning operations with very few consistency-maintenance-required cloning operations. In the aggressive scenario, with our approach, developers may miss a small percentage of cloning operations while still avoid a significant percentage of consistency-maintenance-required cloning operations.
- Adjusting thresholds for different projects may help achieve better effectiveness. It may be desirable to generate a threshold for a certain projects based on the history data of the projects.
- History attributes are not very useful, and the effects of destination attributes vary from Microsoft projects to open source projects. Note that if we do not use history attributes, we may save significant effort on extracting history attributes from historical versions.
- Our approach is less effective on cross-project prediction compared with inner-project prediction, and it is unclear that our cross-project prediction yields results that are effective enough for practical usage. However, cross-project prediction has known to be a very difficult problem in software engineering (due to the variety of software projects), and under most settings, our approach is able to achieve acceptable improvements compared to a random selection on the results based on developers' initial decision.
- There are some differences between the results of our approach on Microsoft subjects and the results on open-source subjects. We will discuss about the differences in detail in Section 6.4.

5.6 Threats to Validity

In our evaluation, we applied our approach to the version histories of four software projects. This factor may be a threat to external validity, since it is possible that our empirical results are specific to the four software projects used in our evaluation and may not be generalizable to other projects. To reduce this threat, we used a subject set including two large Microsoft software projects written in C# and two popular open-source projects written in Java. These projects are from different domains, of different size, written in different programming languages, and developed by different organizations. Although these multi-level differences make it more difficult to explain the differences in results, they help to enhance the representativeness of our experiment, and make the effectiveness of our approach more generalizable.

One major threat to internal validity is the unintentional inconsistent changes (i.e., inconsistency bugs) that may lead us to erroneously mark a cloning operation as consistency-maintenance-free. However, we believe that, for developers, the probability of bringing in inconsistency bugs is much less than the probability of correctly maintaining the consistency. A recent study [22] also support this belief.

A second threat to internal validity is that we are using different code-clone-detection tools in our study on Microsoft software projects and open source software projects. Specifically, the two tools allows different level of changes among code segments (i.e., Microsoft code-clone-detection tool further allows statement insertion/deletion/reorder, on top of identifier-name changes). Such differences may weaken our general conclusion on the four projects, as well as our conclusion on the different results between Microsoft and open source software projects, because it may be the differences in clone definition instead of the differences in code property (industrial or open source) that cause the different results. To reduce this threat, we use two code-clone-detection tools that use relatively similar basic approaches (both transforming statements to token sequences and performing statement hashing). Actually, such differences in code-clone-detection tools may help generalize our conclusions to different code-clone-detection tools, though more thorough studies are required in the future.

A third threat to internal validity is that we use a limited observation time slot to decide whether a code clone experiences divergent changes, which also may lead us to erroneously mark a cloning operation as consistency-maintenance-free. To reduce this threat, we use a relatively long observation time slot, we also did a preliminary study to investigate the possible proportion of erroneous marking (See 6.1).

The main threats to construct validity is that we used cloning operations recovered from the version histories as the training set and the testing set. There might be slight differences between attribute values of recovered cloning operations and attribute values of intended cloning operations because there was some information loss in the version history (e.g., for simultaneously added clone segments, we randomly choose one as the copied piece of code, which may be not the case). However, we believe that this threat to construct validity should not significantly affect the effectiveness of our approach since the resulting differences in attribute values are typically small.

6 DISCUSSION

6.1 Determining the Consistency-Maintenance Status of Code Clones

In our evaluation, we label a code clone as consistency-maintenance-required only if we observe that it experiences consistent changes. So it is important that our observation time period is long enough for potential consistent changes to happen. Otherwise, we may erroneously mark a consistency-maintenance-required code clone as consistency-maintenance-free only because it has not got enough time to experience a consistent change yet. Therefore, we collected cloning operations from only older versions (which had existed in the software projects for more than two years). It should be noted that two years may not be a precise time slot and further investigation may be needed. However, our statistics show that, compared with observing the code clones for only one year, observation of the code clones for the second year, only helps reveal 8 (0.3%) of 2913 once-viewed-as-consistency-maintenance-free cloning operations to become consistency-maintenance-required in XProj, and no once-viewed-as-consistency-maintenance-free cloning operations to become consistency-maintenance-required in the other three projects. So we believe that our labeling should induce very small number of errors.

6.2 Complication of Consistency-Maintenance-Required Clones

Our evaluation shows that our approach is able to achieve a promising effectiveness in two practical scenarios. However, in a number of cloning operations, our approach fails to accurately predict whether they require consistency maintenance. After investigating these cloning operations, we discovered that the main reason why our approach failed in some operations is the complexity of these operations. Generally, these operations tend to have conflicting attributes, in which the values of some attributes indicate the operation to be consistency-maintenance-required and the values of other attributes indicate the operation to be consistency-maintenance-free. Our examples in Section 2 and Section 5.4 have already shown conflicting attributes on concrete code cloning operations. When attributes conflict, it is important to correctly decide the weights of the attributes, which requires the training set to be as large as possible. We believe that involving multiple projects to acquire a larger training set may be helpful in such cases.

6.3 Different Categories of Code Clones

Kapser and Godfrey classified cloning operations into three categories according to their purposes [38]. The three categories are forking clones, templating clones, and customization clones. In forking clones, developers clone a large component for a new environment or different users. In templating clones, developers clone a piece of code elsewhere to perform similar functionalities, such as copying the sorting method from a class for the author list to a class for the paper list. Customization clones are similar to templating clones except that customization clones require revisions after the cloning. The effectiveness of attribute groups in our approach may differ for different categories of clones. For example, for forking clones, history attributes and code attributes may be more important, because in such clones, instability and dependence on components unrelated to the environment are key factors for consistent changes. By contrast, for templating clones, destination attributes may be important since similar contexts will enhance the likelihood of consistent changes. For customization clones, history attributes and code attributes sometimes may be misleading because the revisions in the cloned code affect the precision of these two attributes, while destination attributes may remain discriminative. Therefore, we may further improve our approach by considering Kapser and Godfrey's clone categories, if we are able to automatically identify the category of cloning operations.

6.4 Different Results on Microsoft and Open-source projects

Our experiments on Microsoft and open-source subjects show that, in general, our approach is effective for both Microsoft and open-source subjects. Meanwhile, we also found two main differences between our results on Microsoft subjects and our results on open-source subjects.

- **Destination attributes are not very useful for open source projects.** It should be noted that Microsoft projects are much larger and consider various users and platforms. So these projects usually have quite a lot of modules with similar functionalities for compatibility, or to support different configurations. By contrast, there are much fewer such modules in open source projects. So our destination attributes make more sense in Microsoft projects.
- **Our approach is more effective for Microsoft projects on cross-project prediction.** It should be noted that the two Microsoft projects are both from Microsoft, so that they may share more similarities in software design, programming styles, etc. By contrast, the two open source projects in our subjects are developed by totally different groups, and they should be less similar to each other. This different also indicate that, when performing cross-project prediction, it is important to choose a training project that is similar to the project for prediction.

To sum up, our experiment shows the effectiveness of our approach on both Microsoft and open-source projects, as well as exposes a number of differences in the results of applying our approach on Microsoft and open-source projects. A more in-depth study on more industrial and open-source projects (with comparable sizes, comparable activeness, and from multiple organizations and companies, etc.) in future is required to fully understand these differences.

6.5 Choice of Threshold

Our evaluation shows that the choice of threshold may affect the effectiveness of our approach. In the conservative scenario, although there are slight differences among the results of the four experimental subjects, a threshold of 0.05 works well in general, to allow more than half of code cloning operations with a precision higher than 94%. In the aggressive scenario, the differences among our experimental subjects are larger. For instance, with the threshold of 0.5, the warning rate of our approach varies from 13% to 40%, and the corresponding recall varies from 37% to 72%. The reason for this large difference is that, different projects may differ in the distribution of attributes and the proportion of consistency-maintenance-required code cloning operations, and therefore they will also differ in the distribution of prediction scores.

From Section 5.2.2, and Section 5.3.2, we have a discovery as below. If the warning rate is fixed, the corresponding recall will be more stable (e.g., varying from 47% to 59%, for 20% warning rate). However, since we can only provide a prediction score for each cloning operation, we are only able set a threshold for prediction score, instead of a warning rate. To set better thresholds of our approach for different projects, especially in the aggressive scenario, a possible approach is to use the distribution of prediction scores in the training data to estimate the distribution of prediction scores in the testing data, and set a threshold based on a certain warning rate / recommendation rate for the training data.

For cross-prediction, the choice of threshold becomes more difficult because the distribution of prediction scores in the training data and the testing data may differ significantly. It may be more effective to use similar software projects as training and testing data in cross prediction. Such similarity can be measured by some observable software attributes (e.g., size, domain) and a better threshold may also be derived from the training software projects, but much future work is required to make progress on this problem.

6.6 Different Consistency-Maintenance Requirements

In our paper, for simplicity, we divide the set of cloning operations into only two subsets: consistency-maintenance-required cloning operations, and consistency-maintenance-free cloning operations. Such a division actually deems all consistency-maintenance-required cloning operations as equivalent without considering how much consistency maintenance they require. Table 22 shows the distribution of consistency-maintenance-required cloning operations (in the open-source projects) on the number of consistency changes experienced, and we can see that the number of consistent changes experienced by consistency-maintenance-required cloning operations varies. It is obvious that a cloning operation experiencing only one consistent change in its life time results in less extra maintenance effort compared to a cloning operation experiencing more consistent changes. Therefore, it may be helpful to the developers if we are able to further divide the category consistency-maintenance-required cloning operations to multiple sub-categories with high, medium and low consistency-maintenance requirement.

7 RELATED WORK

In this section, we discuss the previous research efforts that are related to our work.

7.1 Empirical Studies on Code Clones

As far as we know, our paper proposed the first automated approach that predicts whether intended cloning operations require consistency maintenance. Our research is motivated by the findings of recent empirical studies on code clones, and we also use some existing techniques to construct and analyze code clone genealogies. Kim et al. [8] first combined code clone detection tools and version history analysis tools to extract code clone genealogies. Based on clone genealogies, they discovered that it is not always worthwhile to refactor code clones.

Juergens et al. [6] performed an empirical study on the historical code clones of a number of open-source and industry projects. They identified a number of code-clone related software defects, and confirmed that some code clones will cause consistency maintenance. Krinke [7] performed an empirical study on the changes made to code clones. The study shows that the number of consistent changes and inconsistent changes are similar. Kapser and Godfrey also studied code clones in existing software projects and classified clones into categories [38]. Juergens et al. [39] studied a large number of code clones in software projects to find the reasons why developers prefer code clones. Göde and Koschke [21] conducted another empirical study on code clone genealogies. In their study, they discovered that fewer than half of code clones will experience changes and even a smaller proportion will experience consistent changes that lead to consistency maintenance. Thummalapenta et al. [22] performed an empirical study on the evolution patterns of code clones. The major findings of their study include that, 1) in only a small number of cases, developers forget to make consistent changes to code clones, and 2) failing to propagate bug fixes among code clone segments is the main reason for the "forgotten consistent changes." Cai and Kim [23] empirically studied long-lived code clones in software projects, and identified some key attributes in the evolutionary history of a code clone that relate to the existing time of the code clone.

Our research differs from the preceding research in the following two aspects. First, our approach aims to predict whether intended cloning operations require consistency maintenance, while existing research does not provide explicit support for the prediction whether code clones require consistency maintenance. Second, our approach targets the prediction of consistency-maintenance requirement at copy-and-paste time and thus can utilize only attributes available at the time of copy-and-paste, but existing research does not distinguish copy-and-paste attributes from clone evolution attributes and thus can hardly be applied to our problem.

7.2 Code Clone Detection

Code clone detection, which is also closely related to our research, has been a research focus for many years. Generally, code-clone detection techniques fall into three categories: token-based techniques, syntax-tree-based techniques, and semantics-based techniques.

Token-based techniques [25] [26] transform code to token sequences and try to locate code clones by detecting repetitive token sequences. These techniques are the most scalable, but the least robust (can detect identical or near-identical clones but more sensitive to revisions after copy-and-paste). In our research, we used the CloneCodeDetector [32] [12] from Microsoft for code-clone detection in Microsoft projects, and Conqat [13] for the code-clone detection in open-source projects. Both code-clone-detection tools leverage a token-based technique, and they are stable and scalable enough for analyzing large software projects. Syntax-tree-based techniques [9] [27] discovers similar tree structures in the syntax tree of code, and are relatively less scalable, but more robust. Semantics-based techniques [28] [29] [31] use semantic information(e.g., code dependence or behavior) as the base of clone detection, so they are the least scalable, but the most robust.

To compare all these techniques, Bellon et al. [10] did a comparison on a number of existing code-clone detection tools to find the strength and drawbacks of each tool. Based on mutation techniques, Joy and Cordy [11] proposed an automatic framework for comparison of different code-clone detection techniques. On top of code clone detection.

TABLE 22
Break-down of Consistency-Maintenance-Required Cloning Operations on the Number of Consistent Changes

Project	# Consistency-Maintenance-Required Clones	1 Consistent Change	2 Consistent Changes	3+ Consistent Changes
JFreeChart	89	75	12	2
tuxGuitar	60	42	16	2

There are also research efforts on managing detected code clones. Duala-Ekoko and Robillard [5] proposed an approach to track detected code clones during software evolution, so that, when code clones are difficult to remove with refactoring, the developers can maintain these code clones more easily. Later, Nyugen et al., proposed a more powerful clone management tool called Clever [45] [46], which further handles cross-revision clone pairs and uses a tree-mapping algorithm to enhance the precision of tracking clones. Although also trying to relieving the potential consistency-maintenance cost of code clones, compared to our research, these efforts on clone management do not differentiate consistency-maintenance-required and consistency-maintenance-free code clones, and handles only code clones that are already performed.

7.3 Defect Prediction

Another area related to our research is machine-learning-based defect prediction. Defect prediction approaches try to predict the number of defects in a given software component. Similar to our approach, machine-learning-based defect prediction also relies on attributes extracted from code and version histories. Menzies et al. [33] proposed to use multiple classifiers to predict defects and evaluated their techniques on the NASA software defect data. Emam et al. [34] compared different case-based classifiers and concluded that varying the combination of parameters of the classifier does not help to improve prediction precision. Kim et al. [35] further studied the impact of noise in the training data on the effectiveness of defect prediction approaches. Hao et al. [3] proposed an approach to predict whether a runtime error is caused by a defect in the source code or the out-of-date test code. Compared to defect prediction approaches, our approach targets a different problem. Furthermore, our approach uses a different set of attributes. Specifically, among the three attribute groups in our approach, our history attributes are similar to history attributes used in defect prediction; our code attributes focus on code dependence while code attributes in defect prediction focus on code complexity and bad smells; destination attributes are specific to cloning operations.

8 Future Work

In our points of view, the research presented in this paper is only one step towards fully understanding the consistency-maintenance requirement of intended cloning operations. The following directions for further research may help overcome the limitations of our current research.

First, although our evaluation shows that our predictor can provide practical help for developers, there is still large room to improve our approach. In particular, when the prediction score on the consistency-maintenance requirement of a cloning operation is between 0.5 and 0.8, it is still difficult for us to accurately predict whether the operation will be consistency-maintenance-required or consistency-maintenance-free. Furthermore, the result of our experiment shows that some of the attributes we are using may not be suitable for this specific task. We plan to consider adding more attributes, varying existing attributes, and removing non-suitable attributes to improve the effectiveness of our approach. For example, we may use an absolute time threshold instead of a relative time threshold when computing the number of recent changes. Furthermore, besides considering the dependence of the copied code piece, it may also be helpful to consider the maturity and dependence of the code that the copied code piece depends on.

Second, our current evaluation is based on only two software projects from Microsoft and two projects from the open-source community. It would be interesting to evaluate our approach on more software projects, and do cross-project prediction among a suite of software projects. Furthermore, our current evaluation mainly considers two practical scenarios. To evaluate our prediction in more general scenarios, we plan to involve more metrics such as F-score and Cohen's Kappa co-efficient [43] to further evaluate our approach.

Third, our evaluation includes a quantitative study on a number of cloning operations in the version histories of software projects. To fully reveal the strength and weakness of our approach, we plan to design and conduct a number of qualitative case studies involving developers, so that we can further confirm whether the identified consistency-maintenance requirement of cloning operations is consistent with developers' feelings and/or identify more issues for developers to perform cloning operations.

Fourth, currently our tool uses Bayesian Networks as the machine learning algorithm. In the future, we plan to try other major machine learning algorithms (e.g., SVM [40], Decision Trees [41]) and compare the effectiveness of different algorithms.

Finally, our approach can be deemed as an approach for operation-specific defect prediction. Therefore, we plan to do some future work in this direction, such as involving some techniques and attributes in defect prediction to our problem, and / or apply some techniques we have used to predict potential defects resulted from other types of operations. At the same time, the effectiveness of our approach on cross-project prediction is still not desirable, especially on open-source projects. So, in future, we plan to borrow some techniques from existing cross-project defect detection techniques [15].

9 Conclusion

In this paper, we have proposed a novel approach that assists developers in predicting whether consistency maintenance is required for intended cloning operations using Bayesian Networks. Our approach may provide guidance for developers on selectively performing cloning operations in a conservative or an aggressive way with a reduced cost of maintenance. We have also evaluated our approach for both conservative and aggressive scenarios using two large-scale Microsoft software projects, and two popular projects from the open-source community. Our empirical results demonstrate that our approach may be practically useful for both types of scenarios: In the *conservative scenario*, our approach recommend developers to perform more than 50% of cloning operations in which more than 94.0% are truly consistency-maintenance-free. In the *aggressive scenario*, our approach may help developers to avoid 37% to 72% of consistency-maintenance-required cloning operations by warning developers on only 13% to 40% of cloning operations in the four projects.

Acknowledgments

The research is partially sponsored by the National 863 Program of China No. 2013AA01A605, the National 973 Program of China No. 2011CB302604, the Science Fund for Creative Research Groups of China No. 61121063, and the Natural Science Foundation of China No. 91118004, No. 61228203 and No. 61225007.

References

[1] Anvik, J., Hiew, L., Murphy, G. Who should fix this bug? In International Conference on Software Engineering, 361–370, 2006.

[2] Pearl, J. Bayesian Networks: A Model of Self-Activated Memory for Evidential Reasoning, Proceedings of the 7th Conference of the Cognitive Science Society, 329–334, 1988.

[3] Hao, D., Lan, T., Zhang, H., Guo, C., and Zhang, L. Is This a Bug or an Obsolete Test? Proceedings of the 27th European Conference on Object-Oriented Programming, 602–628, 2013

[4] Friedman, N., Geiger, D., and Goldszmidt, M. Bayesian Network Classifiers, Machine Learning, 29(2-3), 131–163, 1997.

[5] Duala-Ekoko, E. and Robillard, M.P. Tracking Code Clones in Evolving Software, In International Conference on Software Engineering, 158–167, 2007.

[6] Juergens, E., Deissenboeck, F., Hummel, B., and Wagner, S. Do code clones matter? In International Conference on Software Engineering, 485–495, 2009.

[7] Krinke, J. A Study of Consistent and Inconsistent Changes to Code Clones, The 14th Working Conference on Reverse Engineering, 170–178, 2007.

[8] Kim, M., Sazawal, V., Notkin, D., and Murphy, G. An empirical study of code clone genealogies, In ACM SIGSOFT Symposium on the Foundations of Software Engineering, 187–196, 2005.

[9] Baxter, I.D., Yahin, A., Moura, L., Sant"Anna, M., and Bier, L. Clone detection using abstract syntax trees, In International Conference on Software Maintenance, 368–377, 1998.

[10] Bellon, S., Koschke, R., Antoniol, G., Krinke, J., and Merlo, E. Comparison and Evaluation of Clone Detection Tools, IEEE Transactions on Software Engineering, 33(9), 577–591, 2007.

[11] Roy, C.K. and Cordy, J.R. A Mutation/Injection-Based Automatic Framework for Evaluating Code Clone Detection Tools In International Conference on Software Testing, Verification and Validation Workshops, 157–166, 2009.

[12] Dang, Y., Zhang, D., Ge, S., Chu, C., Qiu, Y., and Xie, T. XIAO: tuning code clones at hands of engineers in practice, In Proceedings of the 28th Annual Computer Security Applications Conference, 369–378, 2012

[13] Deissenboeck, F., Juergens, E., Hummel, B., Wagner, S., Parareda, B. M., and Pizka, M. Tool Support for Continuous Quality Control, IEEE Software, 25(5), 60–67, 2008.

[14] LaToza, T., Venolia, G., DeLine, R., Maintaining Mental Models: A Study of Developer Work Habits Internation Conference on Software Engineering, 492–501, 2006.

[15] Zimmermann, T., Nagappan, N., Gall, H., Giger, E., and Murphy, B., Cross-project Defect Prediction: A Large Scale Experiment on Data vs. Domain vs. Process, 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering, 91–100, 2009.

[16] Nam, J. and Pan, S. J. and Kim, S., Transfer Defect Learning, International Conference on Software Engineering, 382–391, 2013

[17] Haibo H. and Garcia, E.A., Learning from Imbalanced Data, IEEE Transactions on Knowledge and Data Engineering, 21(9), 1263–1284, 2009.

[18] Nathalie J. and Shaju S. The class imbalance problem: A systematic study Intelligent Data Analysis, 6(5), 429–449, 2002.

[19] Frank, E., Hall, M., Holmes, G., Kirkby, R., Pfahringer, B., Witten, I. H., and Trigg, L. Weka, Data Mining and Knowledge Discovery Handbook, 1305-1314, 2005.

[20] Kohavi, R. A Study of Cross-Validation and Bootstrap for Accuracy Estimation and Model Selection, In International Joint Conference on Artificial Intelligence, 1137–1145, 1995.

[21] Göde, N., Koschke, R. Frequency and risks of changes to clones, In International Conference on Software Engineering, 311–320, 2011.

[22] Thummalapenta, S., Cerulo, L., Aversano, L., and Penta, M. D. An empirical study on the maintenance of source code clones, Emprical Software Engineering, 15(1), 1–34, 2010.

[23] Cai, D., and Kim, M. An Empirical Study of Long-Lived Code Clones, In International Conference on Fundamental Approaches to Software Engineering, 432–446, 2011.

[24] Baker, B. S. On finding Duplication and Near-Duplication in Large Software System, In Working Conference on Reverse Engineering, 86–95, 1995.

[25] Kamiya, T., Kusumoto, S., Inoue, K. CCFinder: a multilinguistic token-based code clone detection system for large scale source code, IEEE Transactions on Software Engineering, 28(7), 654–670, 2002.

[26] Li, Z., Lu, S., Myagmar, S., and Zhou, Y. CP-Miner: finding copy-paste and related bugs in large-scale software code, IEEE Transactions on Software Engineering, 32(3), 176–192, 2006.

[27] Jiang, L., Misherghi, G., Su, Z., and Glondu, S. DECKARD: Scalable and Accurate Tree-Based Detection of Code Clones, In International Conference on Software Engineering, 96–105, 2007.

[28] Gabel, M., Jiang, L., and Su, Z. Scalable detection of semantic clones, In International Conference on Software Engineering, 321–330, 2008.

[29] Wang, X., Lo, D., Cheng, J., Zhang, L., Mei, H., and Yu, J. X.: Matching dependence-related queries in the system dependence graph In IEEE/ACM Conference on Automated Software Engineering, 457–466, 2010.
[30] Navarro, G. A guided tour to approximate string matching, ACM Computing Surveys, 33(1), 31–88, 2001.
[31] Kim, H., Jung, Y., Kim, S., and Yi, K. MeCC: memory comparison-based clone detector, In International Conference on Software Engineering, 301-310, 2011.
[32] Dang, Y., Song, G., Huang, R., and Zhang, D. Code Clone Detection Experience at Microsoft, Proceedings of International Workshop on Software Clones, 63–64, 2011.
[33] Menzies, T., Greenwald, J., and Frank, A. Data Mining Static Code Attributes to Learn Defect Predictors, IEEE Transactions on Software Engineering, 33(1), 2–13, 2007.
[34] Emam, K., Benlarbib, S., Goelb, N., Raic, S. N. Comparing case-based reasoning classifiers for predicting high risk software components, Journal of Systems and Software, 55(3), 301–320, 2001.
[35] Kim, S., Zhang, H., Wu, R., and Gong, L. Dealing with noise in defect prediction, In International Conference on Software Engineering, 481–490, 2011.
[36] Jiang, L., Su, Z., Chiu, E. Context-based detection of clone-related bugs, In ACM SIGSOFT symposium on the Foundations of Software Engineering, 55–64, 2007.
[37] Roy, C. and Cordy, J. An Empirical Study of Function Clones in Open Source Software, In Working Conference on Reverse Engineering, 81–90, 2008.
[38] Kapser, C. and Godfrey M. "Cloning considered harmful" considered harmful: patterns of cloning in software, Empirical Software Engineering, 13(6), 645–692, 2008.
[39] Juergens, E., Deissenboeck, F., Feilkas, M., Hummel, B., Schaetz, B., Wagner, S., Domann, C., Streit, J.: Can Clone Detection Support Quality Assessments of Requirements Specifications? In International Conference on Software Engineering, 79–88, 2010.
[40] Chang, C. and Lin, C.: LIBSVM: A Library for Support Vector Machines, ACM Trans. Intell. Syst. Technol., 2(3), Article No. 27, 2011.
[41] Safavian, S.R. and Landgrebe, D.: A survey of decision tree classifier methodology, IEEE Transactions on Systems, Man and Cybernetics, 21(3), 660–674, 1991.
[42] Cheng, J. and Greiner, R.: Comparing Bayesian Network Classifiers, In Fifteenth Conference on Uncertainty in Artificial Intelligence, 101–108, 1999.
[43] Cohen, J.: Weighted kappa: Nominal scale agreement with provision for scaled disagreement or partial credit, Psychological Bulletin, 70 (4), 213–220, 1968.
[44] Wang, X., Dang, Y., Zhang, L., Zhang, D., Lan, E., Mei, H.: Can I clone this piece of code here? In IEEE/ACM Conference on Automated Software Engineering, 170–179, 2012.
[45] Nguyen, T., Nguyen, H., Pham, N., Al-Kofahi, J., and Nguyen, T.: Clone-Aware Configuration Management, In IEEE/ACM Conference on Automated Software Engineering, 123–134, 2009.
[46] Nguyen, T., Nguyen, H., Pham, N., Al-Kofahi, J., and Nguyen, T.: Clone-Aware Configuration Management, IEEE Transactions on Software Engineering, 38 (5), 1008–1026, 2012.

PLACE PHOTO HERE

Xiaoyin Wang Xiaoyin Wang received his BS degree from the Department of Computer Science at Harbin Institute of Technology in July 2006. In Sept. 2006, he became a PhD student in the School of Electronic Engineering and Computer Science at Peking University, and got his PhD in 2012. After that, he worked in UC Berkeley as a Postdoc for one and half years. In August 2013, he joined the University of Texas at San Antonio as an assistant professor. His research interest is in the area of software engineering, including software maintenance, software mining, and software refactoring.

PLACE PHOTO HERE

Yingnong Dang Yingnong Dang is a principal software development engineer of the Microsoft Azure team. Before joining the Microsoft Azure team in Dec. 2013, he was a Lead Researcher in the Software Analytics group of Microsoft Researcher Asia, Beijing, China. His research interests include software engineering, software analytics, data analytics, and human-computer interaction. Dang has transferred a few important software engineering and data analytics technologies into Microsoft products. In particular, his work on code clone analysis has been integrated into Visual Studio and being used by Microsoft Security Response Center. Dang received a PhD in control science and engineering from the Xian Jiaotong University, Xian, China. He is a member of ACM. Contact him at yidang@microsoft.com.

PLACE PHOTO HERE

Lu Zhang Lu Zhang received the PhD and BS degrees in Computer Science from Peking University in 2000 and 1995, respectively. He is a professor in the Institute of Software, School of Electronics Engineering and Computer Science, Peking University, P.R. China. He was a visiting post-doctoral researcher in School of Computing and Mathematical Sciences, Oxford Brookes University, UK from September 2000 to February 2001. From April 2001 to January 2003, he worked as a post-doctoral researcher in the Department of Computer Science, University of Liverpool, UK. His research interests include testing of software components and component-based software, program comprehension, software maintenance and evolution, software reuse and component-based software development.

PLACE PHOTO HERE

Dongmei Zhang Dr. Dongmei Zhang is a Principal Researcher of Microsoft Research Asia (MSRA). She is also the research manager of the Software Analytics group at MSRA. Her research interests include data-driven software analysis, machine learning, information visualization and large-scale computing platform. She founded the Software Analytics group at MSRA in 2009. Since then she has been leading the group to research and develop innovative data exploration and analysis technologies to help improve the quality of software and services as well as the software development productivity. Her group collaborates closely with multiple product teams in Microsoft, and has developed and deployed software analytics tools which have created high business impacts and successfully been transferred to product teams.

Erica Lan Erica Lan is a partner development manager in Microsoft Azure Compute. She has 18 years experience in development discipline at Microsoft. The products she has been working on include VBA, Access, Knowledge Worker Service, MSN Subscription and Commerce, OneCare, Active Directory Federated System, Forefront online protection for Exchange and Office365 and Bing engineering fundamental. With years of experience as the development manager and a natural inclination for data analysis, Erica has leveraged software engineering analytics techniques for an engineering model. She has successfully used the model to drive the strategic investments in Bing engineering fundamental team and produced the distributed and cache based build system (code name Q) for Bing. Her current passion is to continue using engineering analytics to drive innovations for Microsoft Azure Compute.

PLACE PHOTO HERE

Hong Mei Hong Mei received his BA and MS degrees in computer science from Nanjing University of Aeronautics and Astronautics in 1984 and 1987, respectively; and Ph.D. degree in computer science from Shanghai Jiaotong University in 1992. From 1992 to 1994, he was a postdoctoral research fellow at Peking University. Since 1997, he has been a professor and Ph.D. advisor in the Department of Computer Science and Engineering at Peking University. He has also served as dean of the School of Electronics Engineering and Computer Science and the Capital Development Institute at Peking University, respectively. His current research interests include Software Engineering and Software Engineering Environment, Software Reuse and Software Component Technology, Distributed Object Technology, Software Production Technology, and Programming Language. He is a senior member of the IEEE.