Can I Clone This Piece of Code Here?

Xiaoyin Wang[1]*, Yingnong Dang[2]†, Lu Zhang[1]†, Dongmei Zhang[2], Erica Lan[3], Hong Mei[1]
[1]Key Laboratory of High Confidence Software Technologies (Peking University),
Ministry of Education, Beijing, 100871, China
[2]Microsoft Research Asia, Beijing, China
[3]Microsoft Corporation, One Microsoft Way, Redmond, WA, USA
{wangxy06,zhanglu,meih}@sei.pku.edu.cn, yidang, dongmeiz, erical@microsoft.com

ABSTRACT

While code cloning is a convenient way for developers to reuse existing code, it may potentially lead to negative impacts, such as degrading code quality or increasing maintenance costs. Actually, some cloned code pieces are viewed as harmless since they evolve independently, while some other cloned code pieces are viewed as harmful since they need to be changed consistently, thus incurring extra maintenance costs. Recent studies demonstrate that neither the percentage of harmful code clones nor that of harmless code clones is negligible. To assist developers in leveraging the benefits of harmless code cloning and/or in avoiding the negative impacts of harmful code cloning, we propose a novel approach that automatically predicts the harmfulness of a code cloning operation at the point of performing copy-and-paste. Our insight is that the potential harmfulness of a code cloning operation may relate to some characteristics of the code to be cloned and the characteristics of its context. Based on a number of features extracted from the cloned code and the context of the code cloning operation, we use Bayesian Networks, a machine-learning technique, to predict the harmfulness of an intended code cloning operation. We evaluated our approach on two large-scale industrial software projects under two usage scenarios: 1) approving only cloning operations predicted to be very likely of no harm, and 2) blocking only cloning operations predicted to be very likely of harm. In the first scenario, our approach is able to approve more than 50% cloning operations with a precision higher than 94.9% in both subjects. In the second scenario, our approach is able to avoid more than 48% of the harmful cloning operations by blocking only 15% of the cloning operations for the first subject, and avoid more than 67% of the cloning operations by blocking only 34% of the cloning operations for the second subject.

Keywords

Code cloning, Harmfulness prediction, Bayesian networks, Programming aid

†Corresponding Author

*The work was done when the author was an intern at Microsoft Research Asia.

ASE '12, September 3 – 7, 2012, Essen, Germany

1. INTRODUCTION

Software reuse can reduce the costs of software development. When reusing a piece of code, a developer may have two choices. The first choice is to wrap the reused code into a module (e.g., a method), and invoke the module wherever the code is used. The second choice is to copy the code and paste it in the place where she wants to reuse the code, making some revisions if necessary.

It is typically convenient for developers to reuse existing code with copy-and-paste. A recent study [25] shows that, the schedule pressure often prevents the developers from spending long time on wrapping reused code to modules. Additionally, for some reused piece of code, wrapping them can be technically challenging or require change on the components that developers should not change. As a result, code clones (which are mainly the results of applying software reuse based on copy-and-paste) make up a large proportion of modern software code bases (e.g, more than 20% of the code in Eclipse-JDT are involved in clones or near-clones [23]). However, developers may achieve this convenience at a cost. Maintenance of cloned code requires extra effort, because developers need to consider the consistency among the clone segments when they try to make a revision to one clone segment[1]. Failing to maintain consistency between clone segments may cause bugs [12] [22].

However, recent studies show that code clones are not always detrimental to software development and evolution [24] [7]. In particular, Gode and Koschke [7] demonstrate that less than half of the code clones may change during their life cycle. Among those changes in code clones, only about half are consistent changes (i.e., simultaneous changes of more than one segment in a code clone group), while the others are inconsistent changes (i.e., changes to only one segment without touching the other segments). For inconsistent changes[2], in general developers do not need to consider other clone segments in the clone group, and therefore do not need to make any extra effort when revising a clone segment. For consistent changes, in general developers may have to study all clone segments in the clone group and decide which other segments they should change and how to change them.

The preceding studies indicate that cloning operations can be generally divided into two categories according to whether consistency maintenance is required for the resultant clone segments. The first category of cloning operations, referred to as *harmless cloning operations* in this paper, includes cloning operations whose

[1]In this paper, when several pieces of code are clones to each other, we use the term *clone segment* to represent one of such pieces of code, and we use the term *clone group* to represent a group of clone segments that are clones of each other.

[2]Here we use the term "inconsistent changes" for intentional inconsistent changes. In reality, there may exist unintentional inconsistent changes which lead to bugs, but studies [8] show that such cases are not common.

resultant clone groups are never changed or always changed inconsistently. For this category of code clones, no maintenance of consistency is required, so that developers can perform copy-and-paste and benefit from the convenience for free. The second category of cloning operations, referred to as *harmful cloning operations* in this paper, includes cloning operations whose resultant clone groups need to be changed consistently. They are harmful because any consistent change will incur extra maintenance effort and even lead to code defects if consistency is not maintained. For a harmful cloning operation, wrapping the cloned code into a module should be a better choice. It should be noted that code clones may have other kinds of harmfulness (e.g., bad readability). However, causing consistent changes is one of the most important kinds of harmfulness and has been widely studied. Therefore, we focus on the harmfulness of causing consistent changes in this paper.

Due to various reasons (e.g., lack of expertise or lack of supporting information), developers may have difficulties on wisely determine the harmfulness of a cloning operation wisely. In fact, the statistics on our evaluation subjects (See Table 2) show that the developers of the subjects allow a non-trivial number of harmful cloning operations, which account for about 20% of all cloning operations allowed. Therefore, assistance in understanding the harmfulness of an intended cloning operation may help developers make a better choice for reusing existing code. In this paper, we propose an automatic approach to predicting the harmfulness of intended cloning operations. The intuition of our idea is that some characteristics of the code to be cloned and the context of a code cloning operation may indicate its potential harmfulness. Based on a number of features extracted from the cloned code and the context information of the code cloning operation, we use Bayesian Networks, a machine-learning technique, to predict the harmfulness of an intended code cloning operation. Specifically, we extract three groups of features: historical features that describe the change history related to the code to be cloned, code features that describe syntactical characteristics of the code to be cloned, and destination features that characterize the target place (i.e., the place where the developer intends to paste the code).

To evaluate our approach, we considered two possible usage scenarios of our harmfulness predictor for intended cloning operations: 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 incur harm for maintenance, and the aggressive scenario for radical developers, who want to block a significant proportion of harmful cloning operations while still being able to perform most intended cloning operations. We evaluated our approach using two large-scale software projects of Microsoft in the following way: For the first scenario, we used our approach to approve only cloning operations predicted to be very likely to be harmless; while for the second scenario, we used our approach to block only cloning operations predicted to be very likely to be harmful. Our evaluation results demonstrate that 1) with precision higher than 94.9%, our approach is able to approve 52% to 60% cloning operations for the two projects in the conservative scenario, and 2) by blocking only 15% to 34% of cloning operations for the two projects, our approach is able to avoid 48% to 67% of harmful cloning operations in the aggressive scenario.

The main contributions of the this paper are as follows:

- A demonstration of the feasibility to predict harmfulness of intended cloning operations.
- A harmfulness predictor for cloning operations based on Bayesian Networks using three groups of features that characterize the code to be cloned and the context of the cloning operations.
- A detailed evaluation of the proposed harmfulness predictor using two large-scale industrial software projects under two usage scenarios.

We organize the rest of this paper as below. In Section 2, we provide two examples of harmful and harmless cloning operations. In Section 3, we present our approach in detail. In Section 4, we present an evaluation of our approach using industrial software projects. We discuss some related issues in section 5. We introduce related work in Section 6 and present future works in Section 7. Finally, we conclude this paper in Section 8.

2. EXAMPLES

In this section, we present two examplary cloning operations, which are both from the code base of the first project used in our evaluation. The first example depicts a typical harmful cloning operation and the second example depicts a typical harmless cloning operation. These two examples reveal some clues that harmful and harmless cloning operations may have different characteristics.

In the first example, 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., 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. The first is to change variable `cockpitServers` in Line 3 and other two similar local variables (in Lines 5 and 7) to field variables. The second is to change the constant "PPE" to "BLU" to adapt to the change of naming rules on the server side. The third is to add a function invocation to the configuration fetching part (from Line 2 to Line 7). The fourth is to change the name of `LogLevel.Warning` in Line 21. From this example, we can identify two factors for consistent changes. First, the copy-and-paste is local so that the copied and pasted code pieces share many local variables. Second, the clone segments interacts with the configuration and static fields so that changes on the configuration or static fields may impact both clone segments simultaneously.

Code Snippet 1:

```
1 try{
2   cockpitServers["PPE"] = Config.GetParameter(
3     c_iqmFile,"PPE","cockpitServer",c_ppeCockpitServer);
4   cockpitPorts["PPE"] = Config.GetIntParameter(
5     c_iqmFile,"PPE","cockpitPort",c_ppeCockpitPort);
6   collectionDirs["PPE"] = Config.GetParameter(
7     c_iqmFile,"PPE","collectionDir",c_ppeCollectionDir);
8   if ( Config.GetIntParameter(
9     c_iqmFile,"PPE","rankDataAvailable",
10    c_ppeRankDataAvailable ) == 1 ){
11    m_numCollections++;
12    rankDataAvailable["PPE"] = true;
13  }
14  if ( Config.GetIntParameter(
15    c_iqmFile,"PPE","crawlDataAvailable",
16    c_ppeCrawlDataAvailable ) == 1 ){
17    m_numCollections++;
18    crawlDataAvailable["PPE"] = true;
19  }
20}catch{
21  Logger.Log( LogID.IQM,LogLevel.Warning,
22    "Unable to get cockpitServer or cockpitPort for PPE");
23}
```

Code Snippet 2:

```
1 try{
2   cockpitServers["PROD"] = Config.GetParameter(c_iqmFile,
3     "PROD","cockpitServer",c_productionCockpitServer);
4   cockpitPorts["PROD"] = Config.GetIntParameter(c_iqmFile,
5     "PROD","cockpitPort",c_productionCockpitPort);
6   collectionDirs["PROD"] = Config.GetParameter(c_iqmFile,
7     "PROD","collectionDir",c_productionCollectionDir);
8   if ( Config.GetIntParameter(
9     c_iqmFile,"PROD","rankDataAvailable",
10    c_productionRankDataAvailable ) == 1 ){
11    m_numCollections++;
12    rankDataAvailable["PROD"] = true;
13  }
```

```
14  if ( Config.GetIntParameter(
15    c_iqmFile,"PROD","crawlDataAvailable",
16    c_productionCrawlDataAvailable ) == 1 ){
17    m_numCollections++;
18    crawlDataAvailable["PROD"] = true;
19  }
20}catch{
21  Logger.Log( LogID.IQM,LogLevel.Warning, "Unable
22     to get cockpitServer or cockpitPort for production");
23}
```

In the second example, the developer copied 16 lines of code (depicted below) and pasted it into another method in another file without changes. 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 harmless cloning operations. First, the copied code piece is pasted to another method in another file. Therefore, the resultant 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. However, this piece of code still has some dependence on the other part of the project. For example, `sSource` is a function parameter, which depends on its calling modules.

```
1 if (c >= sSource.Length ||
2   (c == sSource.Length - 1 && sSource[c] == '&')){
3    break;
4 }
5 if (sSource[c] == '&'){
6    try{
7      if (sSource[c+1] == 'q' && sSource[c+2] == '=' &&
8        sTag.ToLower() == "&q="){
9        c += 3;
10     }else{
11       break;
12     }
13   }catch (Exception e){
14     Console.WriteLine(e.Message);
15   }
16}
```

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

3. APPROACH

In this section, we first give an overview of our approach. We then present how we extract features for code cloning operations and construct the predictor based on the historical code cloning operations extracted from version history. We also present the scenarios that our predicting approach can be applied to help developers.

3.1 Overview

The basic idea of our approach is to transform the problem of predicting the harmfulness of intended cloning operations into a problem of learning a prediction model from existing data of performed cloning operations. In particular, we adopt a machine-learning technique named Bayesian Networks. Developed in the late 1980s [2], Bayesian Networks provide a mathematical model 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 for predicting the harmfulness of cloning operations, we need to decide the nodes, the structure, and the weights on the edges in the Bayesian Network. In our problem, the nodes correspond to the events that relate to the harmfulness of cloning operations. To decide the nodes, we collect a number of observable events (i.e., the features described in Section 3.2) that may be related to the harmfulness of intended cloning operations by observing existing cloning operations.

After deciding the nodes in the Bayesian Network, we need to learn the structure and weights of the Bayesian Network from a number of training instances. A training instance is a vector that indicates whether each event happens. The typical algorithm for learning the structure of a Bayesian Network is the K2 algorithm [3], 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 [3], which tries to maximize the probability of the training instances. In our approach, we also rely on these two algorithms[3] to construct the Bayesian Network from a number of cloning operations whose features and harmfulness are already known. In Section 3.3, we present how we determine the features and harmfulness of already performed cloning operations via analyzing the historical versions of software projects.

After constructing the Bayesian Network, we use it to predict the harmfulness of intended cloning operations. In Section 3.4, we present the details of applying the constructed Bayesian Network. We propose two scenarios in Section 3.5, where our predictor may help developers make decisions about code cloning operations.

3.2 Considered Features

As mentioned in Section 3.1, our approach uses a set of features to predict the harmfulness of intended cloning operations. Specifically, we use 21 features in total, which can be divided into three categories: history features, code features, and destination features.

3.2.1 History Features

The reason for using history features is to consider the maturity of the copied piece of code. Intuitively, when a developer perform copy-and-paste, if the copied piece of code is mature and has few bugs, the resultant clone segments will be likely to have few bugs and will not experience many bug-fixing changes in the future. The maturity of a piece of code can be related to the time and the number of changes it experienced. Therefore, we consider the following six history features.

- 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.

[3]Specifically, our approach uses the implementation in Weka [5].

- 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.

Among the six history features, the first three features aim to characterize the changes to the copied piece of code itself, and the other three features aim to characterize the changes to the file containing the copied piece of code.

3.2.2 Code Features

The reason for using code features is to consider the impact of the syntactical characteristics of the copied piece of code on the harmfulness of the cloning operation. 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 will not be very likely to experience changes due to revisions in other parts of the code base. Therefore, we consider the following eight code features.

- 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[4]: Whether the copied piece of code belongs to test code.

The considered code features aim 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 harmfulness beforehand, we consider all these features and rely on the construction algorithm of the Bayesian Network to weight these features.

3.2.3 Destination Features

The reason for using destination features is to consider the similarity between the context of the copied piece of code and the context of the destination of pasting. 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 following seven destination features[5].

- Whether to Be 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 [16]. In all the following features, we also use the 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 feature for file name similarity. For this feature, the value is the same with 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* have *m* parameters (whose names are denoted as P_1, P_2, ..., P_m) and *M2* have *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*)[6]: We define *MPS* to be *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[7]: 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.

Again, when selecting destination features, we only consider whether each feature might be related to the harmfulness of cloning operations. We rely on predictor construction to further sort out the relationships between the features.

3.3 Constructing the Harmfulness Predictor

We use the following three steps to construct our harmfulness predictor. 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 features of the cloning operation and whether the cloning operation is harmful or harmless, thus forming a training instance. Third, we construct the Bayesian Network based on the training instances.

3.3.1 Collecting Existing Cloning Operations

To collect the cloning operations performed in the history of a software project, we first download all the historical versions of the software project and perform clone detection on each of these versions. Then, by mapping code location (paths and file names) of

[4]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.

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

[6]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.

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

the code clones between each version and its previous version [4], we build a clone evolution genealogy. 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.

After building the clone genealogy, 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.

To precisely characterize a cloning operation, we need to determine which clone segment in an original clone group is the copied piece of code and which clone segment contains the pasting destination. Specifically, we use the following two heuristics.

- If one of the clone segment *seg* in an original clone group can be mapped to a code segment in the previous version, we determine that *seg* is the copied piece of code.
- If none of the clone segments can be mapped to a code segment in the previous version, we 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 small effects to our approach because these code segments have exactly the same history features, and usually have similar code features and target features.

Note 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.

3.3.2 Determining Feature Values and Harmfulness of Cloning Operations

To use the collected cloning operations as training instances for constructing our harmfulness predictor, we also need to determine the values of the features and the harmfulness of each collected cloning operation. To determine the values of the features, we analyze the version that the cloning operation is performed on to extract the values of the features for the cloning operation. Since it is common for developers to make multiple changes between two continuous versions and it is difficult to decide the order of the changes, we simply assume that any other changes will not impact the features of the cloning operations. Note that, although our assumption for simplicity may introduce some noise into our training data, the used machine-learning technique can mitigate the impact of the noise in the training process.

As it is difficult to precisely measure the harmfulness of the training instances without human intervention, we just classify the training instances into harmful cloning operations and harmless cloning operations for simplicity. So, the harmfulness of a training instance is either 0 (for harmful) or 1 (for harmless). This simplification may introduce some noise and we rely on the training process to deal with the noise. We automatically determine the harmfulness of a training cloning operation based on the genealogy of the corresponding original clone group using the following heuristics:

- If the clone group experiences no change or only inconsistent changes in the genealogy, we deem the corresponding cloning operation to be harmless.
- If the clone group experiences at least one consistent change, we deem the corresponding cloning operation to be harmful.

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

To check whether and how a clone group changes in its genealogy, we adopt a procedure used in existing empirical studies on code clones [4]. For a 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 inconsistent change.

3.3.3 Training the Predictor

After determining the values of the features and the label of harmfulness for each collected cloning operation, we acquire a set of training instances. Based on the training set, we construct a Bayesian Network, which serves as our harmfulness predictor. Note that the training process automatically deals with noise, irrelevant features, and unorthogonal features.

3.4 Prediction

After constructing the harmfulness predictor, we use it to predict the harmfulness of an intended cloning operation. When a developer intends to perform a cloning operation, our approach extracts the values of its features and uses the trained predictor to predict its harmfulness. Here, our harmfulness predictor provides a prediction score which depicts the probability that the intended cloning operation is harmless. This score can thus help the developers to decide whether to perform the cloning operation. In practice, there can be an application-specific interpretation of the prediction scores to further help developers make the decision.

3.5 Usage Scenarios

As mentioned in Section 1, we propose two scenarios where our approach may help developers make decisions on whether to make a code cloning operation or not.

- *Conservative scenario*. In this scenario, developers are cautious and they only want to perform cloning operations when it is almost certainly safe. They do not want to perform risky code cloning operations.
- *Aggressive scenario*. In this scenario, developers may want to perform as many cloning operations as possible for quick development, due to a tight development schedule or other reasons. Therefore they want to block only a small proportion of the most risky cloning operations and avoid as many harmful operations as possible.

Note that in these two scenarios our predictor provides suggestions to developers when they are going to conduct a code cloning operation. This is a proactive way for preventing harmful cloning

operations. Another possible way to check the harmfulness of a code cloning operation is at the time after it is made but before the cloned code is checked into the version control system. In this way, once a cloning operation is predicted as harmful, 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 operation 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 of further edits for prediction to make our approach more general and applicable to both ways.

4. EVALUATION

In this section, we first introduce the methodology for our evaluation in Section 4.1. Then, we present the evaluation setup in Section 4.2. We present our evaluation results in Sections 4.3, 4.4 and 4.5. We discuss the threats to validity in Section 4.6.

4.1 Methodology

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

Effectiveness for the conservative scenario. In the conservative scenario, we set a threshold (close to 1) and all the cloning operations with a predicted harmfulness value higher than the threshold are deemed as harmless cloning operations. Then we measure the effectiveness of our approach using the two metrics below:

- Approval rate: the proportion of cloning operations that are predicted as harmless in all cloning operations for prediction.
- Precision: the proportion of actually harmless operations in the cloning operations that are predicted as harmless

In this scenario, developers do not want to concede risky cloning operations while they can tolerate some potential harmless cloning operations being blocked, so the precision is expected to be near 100% and the approval rate is not expected to be so high.

Effectiveness for the aggressive scenario. In the aggressive scenario, we set a certain threshold (close to 0) and all the cloning operations with a predicted harmfulness value lower than the threshold are deemed as harmful cloning operations. Then, we measure the effectiveness of our approach using the following two metrics:

- Blocking rate: the proportion of cloning operations that are predicted as harmful in all cloning operations for prediction.
- Recall: the proportion of harmful operations that are predicted to be harmful in all actually harmful operations in the cloning operations for prediction.

In this scenario, since developers pay more attention to performing as many cloning operations as possible and blocking only those cloning operations predicted as harmful with high confidence, it is expected that the recall value should be significantly much higher than the value of the blocking rate.

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

Feasibility for cross-project prediction. Moreover, developers may want to apply our approach on a new project that has a too short 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 harmfulness of the cloning operations in another project. Therefore, we further investigate whether our approach can help when performing cross-project prediction.

Table 1: Subject Software 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

4.2 Evaluation Setup

We carried out our evaluation on two large industrial software projects from Microsoft (denoted as XProj and YProj in this paper, respectively)[8]. We chose the two projects for the following three reasons. First, both of the software projects have large code bases with 1-4 millions lines of code. Therefore, the evaluation results on these two projects may likely be generalizable to typical large industrial software projects. Second, both of the software projects have relatively long version histories, which enable us to extract enough cloning operations and precisely determine the harmfulness of the operations to build our training and testing sets. Third, the two projects belong to different domains and are in different development phases. XProj is a completely new project at Microsoft, launched in 2005, so its version history mainly records the initial development phase. In contrast, YProj is developed based on the code of a previous version of YProj, so its version history mainly records the re-engineering and maintenance phase. Therefore, we can check the effectiveness of our approach on both phases. The information about the two software projects is shown in Table 1. The second column of Table 1 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.

For each software project, we downloaded all the versions (i.e., weekly snapshots) during the life time of the software project. We used weekly snapshots because the numbers of code submissions for these two projects are extremely huge and it is difficult to process them one by one in reasonable time. Then, we extracted cloning operations (using the process described in Section 3.3.1[9]) from all the downloaded versions before December 31st, 2008. We collected cloning operations from only these older versions (which had existed in the software projects for more than two years) because there may not be enough time for us to observe consistent or inconsistent changes in newly generated clone groups. 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, the data between Dec-27-2009 and Dec-27-2010 only helps reveal 8 (0.3%) of 2913 and 0 of 325 once-viewed-as-harmless cloning operations to become harmful in XProj and YProj, respectively. So we believe that our labeling should induce very small error rates. After that, we extracted the harmfulness label and values of the features of all these collected cloning operations (as described in Section 3.3) to build a data set, which provides both the training data and the testing data. Finally, we performed 10-cross validation [6] on the data set to acquire the effectiveness of our approach. In our implementation, we used Weka 3.7 [5] to construct the Bayesian Network and set the weights. Table 2 depicts the details about the data sets that we extracted from the two software projects. In Table 2, columns 1-5 present the project name, the number of versions from which we extracted cloning operations, the number of collected cloning operations, the number of harmful cloning operations, and the number of harmless cloning operations. The numbers in the bracket in

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

[9]In the code clone detection step, we used a near-miss code clone detector from Microsoft [18], with the default setting of the tool, i.e., detecting code clones with no less than 20 lines of code.

Table 2: Details of the Extracted Data Sets

Project	#Versions	#Cloning Operations	#Harmful (%)	#Harmless (%)
XProj	219	3407	502(14.7%)	2905(85.3%)
YProj	117	401	76(19.0%)	325(81.0%)

Table 3: Effectiveness in the *Conservative Scenario*

Project (Threshold)	Approval Rate	Precision
XProj(0.99)	41.3%	96.4%
XProj(0.95)	60.1%	94.9%
XProj(0.90)	67.0%	93.9%
XProj(0.85)	68.7%	93.7%
XProj(0.80)	71.0%	93.2%
YProj(0.99)	46.6%	96.3%
YProj(0.95)	51.9%	95.7%
YProj(0.90)	54.4%	94.0%
YProj(0.85)	55.4%	94.1%
YProj(0.80)	56.4%	93.4%

columns 4-5 present the proportion of harmful and harmless operations in all cloning operations, respectively.

From Table 2, we have two observations. First, in both software projects, the numbers of harmless cloning operations are much greater than the numbers of harmful cloning operations. This observation is consistent with the findings of the empirical study by Gode and Koschke [7]. Second, YProj has fewer cloning operations compared with XProj. The reason is that YProj is in the re-engineering phase, so there is less newly added code and thus fewer cloning operations. Note that we did not collect cloning operations from code clones that exist in the initial version of YProj, because in such a case we were unable to accurately determine the harmfulness or the values of the features of those cloning operations.

4.3 Effectiveness in the Two Scenarios

Table 3 depicts the effectiveness of our approach for the *conservative scenario* using the following threshold values: 0.8, 0.85, 0.9, 0.95, and 0.99. In Table 3, column 1 presents the combination of the project and the threshold value, column 2 presents the approval rate, and column 3 presents the precision of our approach.

From Table 3, we have the following observations. First, with the threshold value of 0.95, our approach is able to predict about 50% to 60% of the cloning operations as harmless 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 safe while still allowing more than 50% of the cloning operations. Without our approach, it is difficult for developers to make such decisions with confidence. For example, we can treat all the 3407 cloning operations in proj X as "approved" cloning operations based on the judgement of developers of proj X, since they have been already checked in to the version control system of proj X. However, as shown in Table 3, the precision of "prediction" by developers is 85.7%, which is much lower than the precision of our approach. Therefore, our approach provides conservative developers a guidance for approving only harmless cloning operations with high precision.

Table 4 depicts the effectiveness of our approach for the *aggressive scenario* using the following threshold values: 0.1, 0.2, 0.3, 0.4, and 0.5. In Table 4, column 1 presents the combination of the project and the threshold value, column 2 presents the blocking rate, and column 3 presents the recall.

From Table 4, we have the following two observations. First, with all the threshold values, our approach is able to avoid a large proportion of harmful cloning operations by blocking a much smaller proportion of cloning operations. For example, with the threshold value of 0.3, our approach is able to avoid 48.6% of the harmful operations in XProj by blocking only 14.5% of all cloning operations, and avoid 67.1% of the harmful operations in YProj by blocking only 34.4% of all cloning operations. This demonstrates the value of our approach in the aggressive scenario: With our approach, developers may miss a small percentage of cloning operations while still blocking a significant percentage of harmful cloning operations. Without our approach, the blocking rate and the recall should be about the same, as the data used in our evaluation already reflects the practice of developers to perform cloning operations. Second, the experimental results for the two projects have different blocking rate for a same threshold. This indicates that a different threshold may be used for a different projects to better utilize our approach.

Table 4: Effectiveness in the *Aggressive Scenario*

Project (Threshold)	Blocking Rate	Recall
XProj(0.1)	7.3%	33.9%
XProj(0.2)	11.9%	45.0%
XProj(0.3)	14.5%	48.6%
XProj(0.4)	18.1%	57.0%
XProj(0.5)	19.7%	59.0%
YProj(0.1)	23.2%	55.3%
YProj(0.2)	30.7%	64.5%
YProj(0.3)	34.4%	67.1%
YProj(0.4)	38.2%	69.7%
YProj(0.5)	39.9%	72.4%

Table 5: Precision of the Three Variants in the *Conservative Scenario*

Project (Approval 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%

4.4 Impacts of Feature Groups

As our approach uses three groups of features to predict harmfulness of cloning operations, we removed each group of features at one time to check how each group of features contribute to the overall effectiveness of our approach. We experimented with three variants of our approach, each using two groups of features. 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 can avoid 10% of harmful cloning operations by blocking 5% of cloning operations, while variant B can avoid 20% of harmful cloning operations by blocking 10% of cloning operations. It would be difficult to judge which variant is better. Therefore, in the comparison, we fixed the approval rate as 50%, 60%, and 70% (i.e., in the approval rate range of our approach using thresholds between 0.8 and 0.99) for the *conservative scenario* and fixed the blocking rate as 10%[10], 20%, and 30% (i.e., in the blocking rate range of our approach using thresholds between 0.1 and 0.5) for the *aggressive scenario*. Tables 5 and 6 depict the results of comparing the three variants together with our approach (i.e., using all three groups of features) for the two projects.

From Tables 5 and 6, we have the following observations. First,

[10]Note that blocking 10% cloning operations is equivalent to approving 90% cloning operations.

Table 6: Recall of the Three Variants in the *Aggressive Scenario*

Project (Blocking 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%

Table 7: Effectiveness for Cross-Project Prediction in the *Conservative Scenario*

Setting(Threshold)	Approval Rate	Precision
XProj-YProj(0.99)	52.4%	93.8%
XProj-YProj(0.95)	61.6%	91.5%
XProj-YProj(0.90)	66.3%	90.2%
XProj-YProj(0.85)	67.8%	89.3%
XProj-YProj(0.80)	70.3%	88.7%
YProj-XProj(0.99)	22.5%	95.3%
YProj-XProj(0.95)	33.1%	94.1%
YProj-XProj(0.90)	40.8%	94.4%
YProj-XProj(0.85)	41.2%	94.4%
YProj-XProj(0.80)	43.6%	94.3%

removing the history features has small impacts on the prediction in both scenarios. Second, removing the destination features results in significantly negative impacts on the effectiveness in both scenarios. Third, removing the code features results in a small impact in the *conservative scenario* but a significantly negative impact in the *aggressive scenario*. Fourth, removing some features may even slightly improve the results in some scenarios due to possible noise in those features. These observations indicate that it may be feasible to use only the code features and the destination features to predict harmfulness of intended cloning operations.

4.5 Effectiveness of Cross-Project Prediction

In Section 4.3, 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 referred to as cross-project prediction. Cross-project prediction is notoriously difficult for many mining based software engineering approaches [1], since software projects are often largely 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 difference in various aspects. Therefore, it would be interesting to study whether our approach still provides some help for cross-project prediction.

To perform cross-project prediction, we evaluated our approach by training our predictor with the data from XProj and test the predictor on YProj, and vice versa. Tables 7 and 8 depict the results of cross-project prediction in the two scenarios, respectively. In both tables, we use XProj-YProj to denote training on XProj and testing on YProj, and YProj-XProj to denote the opposite.

From Table 7, we can observe that, with the threshold 0.95, for the setting of XProj-YProj our approach can approve 60% cloning operations with a precision of 91.5%, and for the setting of YProj-XProj, our approach can approve 33.1% cloning operations with a precision of 94.1%. Compared with Table 3, we can observe that

Table 8: Effectiveness for Cross-Project Prediction in the *Aggressive Scenario*

Setting(Threshold)	Blocking Rate	Recall
XProj-YProj(0.1)	0.5%	1.3%
XProj-YProj(0.2)	3.0%	3.9%
XProj-YProj(0.3)	4.0%	5.3%
XProj-YProj(0.4)	5.5%	7.9%
XProj-YProj(0.5)	16.5%	32.9%
YProj-XProj(0.1)	22.9%	52.3%
YProj-XProj(0.2)	33.3%	67.3%
YProj-XProj(0.3)	34.0%	68.1%
YProj-XProj(0.4)	42.3%	72.1%
YProj-XProj(0.5)	45.7%	77.9%

the effectiveness of our approach on cross-prediction drops. However, in YProj, the proportion of harmless 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, compared to random selection (note that the improvement is not trivial considering that the precision is already very high). Furthermore, with the threshold 0.99, our approach with the XProj-YProj setting may approve more than 50% of cloning operations with a precision of 93.8%, and with the threshold 0.8, our approach with the YProj-XProj setting may approve more than 43% of cloning operations with a precision of 94.3%. This observation demonstrates that our approach has the potential to be used in cross-project prediction in practice if developers can appropriately tune the threshold to keep an appropriate blocking rate.

Similarly, comparing Table 8 and Table 4, 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 threshold 0.3, for the setting of XProj-YProj, our approach blocks 4% cloning operations to avoid 5.4% of harmful cloning operations, and for the setting of YProj-XProj our approach blocks 34.0% to avoid 68.1% harmful cloning operations. However, for the setting of YProj-XProj, our approach still achieves a 100% improvement over random selection, and for the setting of XProj-YProj, our approach may achieve a similar improvement if the threshold is appropriately tuned.

In summary, empirical results presented in this subsection indicate that our approach is able to provide some help for cross-project prediction, and have a large space of enhancement if developers can appropriately tune the threshold. Furthermore, it should be noted that, as we used two quite different projects in our evaluation, our cross-prediction had to be based on training on one project and testing on the other project and thus concede significant negative impacts induced by project differences. In practice, we may use multiple projects similar to the target project for training so that the trained predictor is more suitable for the target project. Furthermore, as our approach is based on machine learning, it is also possible to gradually add cloning operations performed in the target project into the training set to make the training process adaptable to the target project. We expect these enhancements to further boost our approach for cross-project prediction.

4.6 Threats to Validity

In our evaluation, we applied our approach to the version histories of two software projects. This factor may be a threat to external validity, since it is possible that our empirical results are specific to the two software projects used in our evaluation and may not be generalizable to other projects. To reduce this threat, we chose large industrial software projects as subjects from different domains. The main threats to internal validity is the unintentional

inconsistent changes (i.e., inconsistency bugs) which may lead us to erroneously mark a cloning operation as harmless. However, we believe that, for developers, the probability of bringing in inconsistency bugs is much less than the probability of correctly maintaining the consistency. Recent studies [8] also support this belief. Another threats to internal validity is that we use a limited observation time slot to decide whether a code clone experiences inconsistent changes, which also may lead us to erroneously mark a cloning operation as harmless. To reduce this threat, we use a relatively long observation time slot (two years), and our study in Section 4.2 shows that such long time slot will bring in few errors in marking cloning operations. 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 feature values of recovered cloning operations and feature 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 have drastic impacts on the effectiveness of our approach since the resulting differences in feature values are typically small.

5. DISCUSSION

Kasper and Godfrey classified cloning operations into three categories according to their purposes [24]. 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 functions, 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. Actually, the effectiveness of feature groups in our approach may differ for different categories of clones. For example, for forking clones, history features and code features 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 features may be important since similar contexts will enhance the likelihood of consistent changes. For customization clones, history features and code features sometimes may be misleading because the revisions in the cloned code affect the precision of these two features, while destination features may remain discriminative. Therefore, we may further improve our approach by considering Kasper and Godfrey's clone categories, if we are able to automatically identify the category of cloning operations.

6. RELATED WORK

To the best of our knowledge, the research presented in this paper is the first automatic approach that predicts the harmfulness of intended cloning operations. Our research is motivated by the findings of recent empirical studies on code clones, and we also use some existing techniques to process code clone genealogies. Kim et al. [4] 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. Kapser and Godfrey also studied code clones in existing software projects and classified clones into categories [24]. Juergens et al. studied a large number of code clones in software projects to find the reasons why developers prefer code clones [25]. Gode and Koschke [7] reported another empirical study on code clone genealogies. In their study, they discovered that only less than half of code clones will experience changes and even a smaller proportion will experience consistent changes that lead to extra maintenance cost. Thummalapenta et al. [8] performed an empirical study on the evolution patterns of code clones. The major findings of their study include 1) that in only a small number of cases, developers forget to make consistent changes to code clones, and 2) that failing to propagate bug fixes among code clone segments is the main reason for the "forgotten consistent changes". Cai and Kim [9] empirically studied long-lived code clones in software projects, and identified some key features 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 harmfulness of intended cloning operations, while existing research does not provide explicit support for harmfulness prediction for code clones. Second, our approach targets harmfulness prediction at copy-and-paste time and thus can utilize only features available at copy-and-paste time, but existing research does not distinguish copy-and-paste features from clone evolution features and thus can hardly be applied to our problem.

Code clone detection, which is also closely related to our research, has been a research focus for many years. Due to space limit, we list only some of representative research in the area of code clone detection. Kamiya et al. [11] developed CCFinder, which transforms a program to tokens and detect clones by performing token-by-token comparison. Li et al. [12] proposed CP-Miner, which uses frequent sequence mining to identify similar sequences in the tokenized program. Jiang et al. [13] proposed Deckard, a syntax tree-based code clone detection tool, which discovers similar tree structures in the syntax tree of the code base. Later, researchers also developed approaches to find similar structures in system dependence graph of a software [14] [15]. Gable et al. [14] simplified system dependence graphs of a software code base to trees and use an algorithm similar to Deckard to detect similar dependence structures. Recently, Kim et al. [17] proposed an approach based on symbolic execution to detect semantic clones in the code base. In our research, we used the CloneCodeDetector [18] from Microsoft for data collection in our evaluation, because CloneCodeDetector is scalable and stable enough to be efficiently applied in more than 100 versions of the huge code bases of the two projects used in our evaluation.

Another research 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 features extracted from code and version histories. Menzies et al. [19] proposed using multiple classifiers to predict defects and evaluated their techniques on the NASA software defect data. Emam et al. [20] compared different case-based classifiers and concluded that varying combination of parameters of the classifier does not help to improve prediction precision. Kim et al. [21] further studied the impact of noise in the training data on the effectiveness of defect prediction approaches. Compared to defect prediction approaches, our approach targets at a different problem. Furthermore, our approach uses a different set of features. Specifically, among the three feature groups in our approach, our history features are similar to history features used in defect prediction; our code features focus on code dependence while code features in defect prediction focus on code complexity and bad smells; destination features are specific to cloning operations.

7. FUTURE WORK

We deem the research presented in this paper as the first step towards fully understanding the harmfulness of intended cloning

operations. The following directions for further research may help overcome the limitations of our current research.

First, although our evaluation indicates that our harmfulness predictor can provide practical help for developers, there is still a large space to improve our approach. In fact, when the prediction score on the harmfulness of a cloning operation is between 0.5 and 0.8, it is still difficult for us to accurately predict whether the operation to be harmful or harmless. We plan to consider adding more features and/or varying existing features to improve the effectiveness of our approach. For example, we may use absolute time threshold instead of 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 Microsoft software projects written in C#. It would be interesting to evaluate our approach on more software projects, such as open source software projects and/or software projects written in other languages. 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 [26] to further evaluate our approach.