Machine Learning Model for Detecting Code Clone Related Bugs

Chowdhury Ashabul Yeameen
The University of Texas, Dallas
cxy111130@utdallas.edu

Seungtak Baek
The University of Texas, Dallas
sxb085000@utdallas.edu

Abstract

Though intentionally copy-pasted code, aka clone code, are sometimes considered as the replication of tested code, they can also lead to a difficult to maintain codebase because there are multiple places to update for single modification. Failed to update all the places, which is very common, results in inconsistent clones. Sometimes asymmetric modifications happen just because that part needed some added/modified functionality. But many other times those changes are intended to fix a bug in it. And since the link between clones are usually not maintained, that modification does not propagates and that particular bug remains alive on its clone(s).

1. Introduction

Intentional code cloning, otherwise known as copy-paste of codes, is considered as a bad practice in Software Engineering. But sometimes there are no alternatives other than cloning when some code is used as a template or sequence of function call is necessary for achieving some functionality. Example of these cases are hardware drivers where people use some kind of template to start with and resource allocation/deallocation methods which are used sequentially and repeated over and over in many places in a software code. Some recent studies found its unavoidable as clone code exist in all major large scale software [1, 6, 8]. The study by *Kamiya et al.* [6] found that 12 percent of Linux code was actually duplicate whereas *Baker et al.* [1] showed 19 percent of X Window System showed the sign of duplications. A study of *Code Clone Genealogy* [7] showed even active refactoring followed by many agile methodologies is not enough to remove all the clone codes.

There are many techniques for suggested for detecting exact and near match code clones. *I. Baxter et al.* [2] and *Lingxiao Jiang et al.* [5] used Abstract Syntax Tree, Zhenmin Li et al [5] used frequent subsequence mining of tokens whereas *Kamiya et al.* [6] used sequence of tokens for detecting code clones.

There are some recent researches on detecting clone related bugs. But all of these approaches are based on some heuristics and/or rule based techniques. *Deckard* [5] considers difference of contexts in the clones. The study by *Dawson Engler et al* [3] goes even further to find anomaly in frequently used template code as a sign of bug and come up with decent accuracy, and more importantly, ranking of possibility of those bugs in OpenBSD code base. *Mark Gabel et al.* [4] proposed a scalable tool *DejaVu*, an independent and extended implementation of *DECKARD* to detect bug in a 75+ million lines of code base and achieved 30 percent accuracy. Though applying many such heuristics and relaxed matching improve recall, all the techniques heavily suffers from accuracy or recall. We took off where other methods left off - get near match clones by using existing techniques, extract features, and apply machine learning techniques to improve accuracy. After applying machine learning technique, accuracy improved more than 2-fold to any existing work with over 80% on the sample case of finding bug in Firefox codebase.

2. Problem Definition and System Overview

2.1 Task Definition

The goal of this project is to build a effective machine learning model to detect harmful code in software by observing nearly identical code clones. The bug type does not include the sign of bad practices due to intentional copy/paste; there are lot of tools available for that purpose. And the model works only on clone pairs when one is slightly modified and not propagated to other one. It can detect two specific types of bugs:

1. When there was a bug in intentionally copy/pasted code clone and one copy got the fix but that fix didnt propagated to other copy.
2. A new bug introduced during copy-paste time while changing the code for adopting the contextual differences from source to destination.

Figure 1 showed an example of confirmed bug detected in Mozilla Firefox code. The bug is function returns true without checking the len value (line 24). Without getting knowledge about the domain it is impossible to detect the bug.

Original code duplicated and pasted in the same file about 100 lines distance. Figure 2 shows the duplicated code. But in this part someone fixed the bug and ensured the value of len is not zero before returning true. The original bug remained possibly because the fixer doesnt know that there is a duplicate.

There are several initiative to detect these kind of bugs in software clone anomalies and all of these showed accuracy between 10%-30% accuracy. Those techniques[1,3,4,5,7,8,9,10]

```
// File: mozilla\content\canvas\src\webglcontextgl.cpp 1095:14 -> 1120:4

if (!GetGLName<WebGLProgram>("getActiveAttrib: program", pobj, &progname))
        return NS_OK;

    NativeJSContext js;
    if (NS_FAILED(js.error))
        return js.error;

    MakeContextCurrent();

    GLint len = 0;
    gl->fGetProgramiv(progname, LOCAL_GL_ACTIVE_ATTRIBUTE_MAX_LENGTH, &len);
    if (len == 0) {
        *retval = nsnull;
        return NS_OK;
    }

    nsAutoArrayPtr<char> name(new char[len]);
    PRInt32 attrsize = 0;
    PRUint32 attrtype = 0;

    gl->fGetActiveAttrib(progname, index, len, &len, (GLint*) &attrsize, (WebGLuint*) &attrtype, name);
    if (attrsize == 0 || attrtype == 0) {
        *retval = nsnull;
        return NS_OK;
    }
```

Figure 1. Buggy copy of the clone

```
// File: mozilla\content\canvas\src\webglcontextgl.cpp 1146:14 -> 1172:4

  if (!GetGLName<WebGLProgram>("getActiveUniform: program", pobj, &progname))
        return NS_OK;

    NativeJSContext js;
    if (NS_FAILED(js.error))
        return js.error;

    MakeContextCurrent();

    GLint len = 0;
    gl->fGetProgramiv(progname, LOCAL_GL_ACTIVE_UNIFORM_MAX_LENGTH, &len);
    if (len == 0) {
        *retval = nsnull;
        return NS_OK;
    }

    nsAutoArrayPtr<char> name(new char[len + 3]); // +3 because we might have to append "[0]", see below
    PRInt32 attrsize = 0;
    PRUint32 attrtype = 0;

    gl->fGetActiveUniform(progname, index, len, &len, (GLint*) &attrsize, (WebGLenum*) &attrtype, name);
    if (len == 0 || attrsize == 0 || attrtype == 0) {
        *retval = nsnull;
        return NS_OK;
    }
```

Figure 2. Fixed copy of the clone

start with near match clone detection and applied some rules to distinguish bugs from benign code. Those rule based solutions are based on some kind of heuristics which are sometimes very much target project specific and suffer from platform dependency, or in many cases project dependency. The target of this machine learning model is provide a generic solution for detecting this particular type of bugs without any manual training and to prove this can lead to some significant improvement in accuracy over any other existing work.

2.2 System Overview

Our proposed Learning method requires labeled data to work. We started with a sample output of recent clone detection research[10] published in OPSLA 2010 and improve the accuracy on the same data. The procedure involves the following steps:

1. **Extract code features from the nearly match clones diff output**
For this particular problem features are the semantic difference between code clones. Because of the strict control on intolerance of variance between clones, there doesnt exist much difference between code clones. Most of the times difference is by modification of few tokens. Therefore for each data point there are generally one or two feature. One alternative approach can be considering one specific feature and map to multiple generalize feature. For example assigning a value to a new variable can be considered as both new def-clear path as well as variable declaration. But most cases this didnt show any significant difference in result that's why such mapping is avoided in this project. Those features are extracted and annotated manually in an spreadsheet and later a script read that file and generated arff.
2. **Extract few more features like distance between code clones**
If both clones exist within one or few screen distance (considering 80 lines as a single screen) most likely it is maintained by the same developer or one has the knowledge of existence of the clone. But after few screen of distance it is almost same. For example 20 screen distance has the same effect as 200 screen distance. So these values are normalized to a scale between 0 and 1 where few lines of distance is considered as 0 and beyond 200 lines of distance is considered as 1. Number of lines between these two are normalized to value between 0 and 1. Another feature we have considered whether clones exist in the same file or in some different file. This step is done completely automatic and merged with other features for generating arff by the script.
3. **Minimize the number of classes**
In the original data, there are 5 classes based on the severity of the harmfulness. With that number of classification and lower number of training made the the model suffer. So we have reduced the number of classification to 2 from 5. But the boundary between these two was flexible and we tried with different segmentation of the original class. This is important since the requirement of the software project can determine whether they will only look for certain bugs or close smell is as important as bugs.
4. **Try with different machine learning techniques**
To determine the best algorithm without suffering from overfitting we tried with several algorithms. Since there are very few data point with more than one syntax feature and distance and cross file are completely independent of code, we can assume features are conditionally independent. we tried with Naive Bayes, SVM, and Neural Network for classification.
5. **Effectiveness Testing**
Since we have very limited number of data, we have applied 10-fold operation in weka to both train and test on the same set of data.

3. Experimental Evaluation

3.1 Data

The input arff file is generated from manually mined feature matrix and the properties of the diff output of DejaVu[10]. The feature matrix is generated manually by observing diff output. Feature matrix has five different classes which are defined as:

- BUG Clear sign of bug
- CHECK Possible bug; requires domain knowledge to verify
- SMELL Code smell. Not good practice and may end up with introducing bug
- STYLE Change of style, say adding braces for single statement if clause
- FALSE Most likely not bug

These five different classes are reduced to two classes by our script. We choose binary classification as there are very little difference between some of those five classes and we needed more data to train our system which are not available for te time being. The script is configurable to generate arff file with custom classification boundary; we can define which of source classes to merge into one.

3.2 Result

Dataset 1
HARMFUL=BUG, CHECK, SMELL (96 CASES)
BENIGN=STYLE, FALSE (34 CASES)

Both Bugs and Checks are highly probably code bug where Smells may lead to introducing bug or considered bad practices. So all of these need developer attention to be fixed. On the other hand, Style and False cases are harmless. If we see the result in Table 1, there is very high accuracy in detecting Harmful code where little less in Benign. Thats a good sign since we also need high recall in detecting Harmful code.

Dataset 2
HARMFUL=BUG, CHECK, SMELL, STYLE (106 CASES)
BENIGN=FALSE (24 CASES)

In this case Style is added to Harmful class which may not add much value but very high recall, particularly in one case we got 100%, is probably most desirable for any software project. But this one suffers from low accuracy on detecting False cases.

Dataset 3
HARMFUL=BUG, CHECK (70 CASES)
BENIGN=FALSE (24 CASES)

This dataset selectively considers only three of the original classification: Bug, Check in one side and False on other. Other two classes, Smell and Style are discarded while building the dataset.
Bug and Check shows some very strong sign of bug and need to address as soon as possible where other cases can

Powers	Meta	Parameter	Harmful			Benign			Overall
			Correct	Missed	Recall	Correct	Missed	Recall	
SVM		G=0.2	92	4	95.8%	8	26	23.5%	76.92%
SVM	AdaBoost	G=.2,I=30	83	13	86.5%	17	17	50%	76.92%
SVM	Bagging	G=.2,I=30	92	4	95.8%	8	26	23.5%	76.92%
NB			83	13	86.5%	22	12	64.7%	80.77%
NB	Bagging	I=30	85	11	88.5%	21	13	61.8%	81.53%

Table 1. Effectiveness on Recalling Bug, Check and Smell

Powers	Meta	Parameter	Harmful			Benign			Overall
			Correct	Missed	Recall	Correct	Missed	Recall	
NB			98	8	92.5%	17	7	70.8%	88.46%
NB	AdaBoost	I=30,P=100	98	8	92.5%	16	8	66.7%	87.69%
NB	AdaBoost	I=30,P=50	98	8	92.5%	18	6	75%	89.23%
SVM		G=.2	106	0	100%	7	17	29.2%	86.92%
SVM	AdaBoost	G=.2,I=30	100	6	94.3%	15	9	62.5%	88.46%
SVM	AdaBoost	G=.2,P=50	100	6	94.3%	16	8	66.7%	89.23%
SVM	Bagging	I=10	105	1	99.1%	9	15	37.5%	87.7%

Table 2. Effectiveness on Reducing False Cases

Powers	Meta	Parameter	Harmful			Benign			Overall
			Correct	Missed	Recall	Correct	Missed	Recall	
NB			65	5	92.9%	21	3	87.5%	91.49%

Table 3. Effectiveness on Selective Classification

be easily ignored for the time being. This classification may not be practical since we are not taking two whole class of data which is not possible to do automatically. Still one test case is given here just to show how effective the model is to detect Buggy cases

3.3 Discussion

Our goal for this project was not only to increase classification accuracy, but to reduce the number of true negative in detecting clone related bugs. We started with a maximum accurate data and applied our method to reduce false positive cases. We are able to show promising output in retaining above 90% of Harmful cases while reducing benign data points significantly. We are hopeful that larger set of labeled data and little more relaxed input will yield even higher output.

4. Threat to Validity

One big threat to our work is that we included smell and some other bad practices together with real bugs in our classification which may not be the goal of bug finding. But our dataset is so small that classifying detected real bug may not produce any dependable model. Again finding Check and Smell is also important in most of the projects where developers try to eliminate those as early as possible before going to production. Our model did significantly good in reducing the number of cases developers need to check. In other words, this will refine the result of existing method with very high certainty of finding bad code.

Another threat can be the specific dataset we have chosen to use. We admit that we worked on a single project and a very small dataset. But here we just wanted to emphasis on our goal that even better result can be produced by applying machine learning along with other bug detection techniques. And larger and wide variety of project dataset are suppose to increase the chance to learn more and work better rather than decreasing accuracy.

5. Future Work

Possibly the most important task is to automate feature extraction from clone data. So far we have done this manually because without getting enough lexical information we cannot do this automatically. Getting lexical information will not be a big problem and we are hoping to come up with a complete automated feature extractor in our next work.

The very next important step is to try on a bigger dataset. We already produced a large set of nearly identical clones by using DECKARD on latest linux kernel 3.4rc4. But verifying the data will take a lot of effort and even we cannot confirm a bug by ourselves without the approval of the project developers.

After training with larger set of data we may do a study on the kind of semantic changes show the sign of bug fixing. We may come up with a tool that can automatically detect the changes related to bug fix even when that are not annotated/linked with a corresponding issue and warn the developers before checking in their code.

6. Conclusion

Discovering software bug automatically is a highly researched area. Even after a lot of effort in writing test cases it is nearly impossible to verify a software as completely free of bugs. Also high false positive keeps developers away from many of the automatic bug detection techniques. We choose a particular field of automatic bug detection and showed a significant improvement can be made by using machine learning technique on important extracted features. Moreover, the process will be completely automatic and promise to save developers$'$ time to find out the bug.

References

[1] B. S. Baker. On finding duplication and near-duplication in large software systems. pages 86–95, 1995.

[2] Y. A. M. L. S. M. B. L. Baxter, Ira D. Clone detection using abstract syntax trees. pages 368–377, 1998.

[3] C. D. H. S. C. A. C. B. Engler, D. Bugs as deviant behavior: A general approach to inferring errors in systems code. volume 35, pages 57–72, 2001.

[4] Y. J. Y. Y. G. M. S. Z. Gabel, M. Scalable and systematic detection of buggy inconsistencies in source code. pages 175–190, 2010.

[5] M. G. S. Z. G. S. Jiang, L. Deckard: Scalable and accurate tree-based detection of code clones. pages 96–105, 2007.

[6] K. S. I. K. Kamiya, T. Ccfinder: A multilinguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering*, 28(7):654–670, 2002.

[7] S. V. N. D. M. G. Kim, M. An empirical study of code clone genealogies. pages 187–196, 2005.

[8] L. S. M. S. Z. Y. Li, Z. Cp-miner: Finding copy-paste and related bugs in large-scale software code. *IEEE Transactions on Software Engineering*, 32(3):176–192, 2006.