Machine Learning Model for Detecting Clone Related Bugs

Chowdhury Ashabul Yeameen
cxy111130@utdallas.edu

1 Introduction

Though duplicate 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).

Intentional code cloning, otherwise known as copy/paste of codes, is considered as 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][5][7]. The study by Kamiya et al[7] found that 12 percent of Linux code was actually duplicate whereas Baker[7] showed 19 percent of X Window System showed the sign of duplications. A study of code clone genealogy[2] 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. Maxter et al[3] and Lingxiao Jiang et al[4] used Abstract Syntax Tree, Zhenmin Li et al [5] used frequent subsequence mining of tokens whereas Kamiya et al[5] 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[4] considers difference of contexts in the clones whereas PR-Miner[8] considers anomalies in code duplicates as a sign of bug. The study by Dawson Engler et al[9] 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[10]proposed a scalable tool DejaVu to detect bug in a 75+ million lines of code base and got 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

```
1: // File: mozilla\content\canvas\src\webglcontextgl.cpp 1095:14 -> 1120:4
2:
3: if (!GetGLName<WebGLProgram>("getActiveAttrib: program", pobj, &progname))
4:         return NS_OK;
5:
6:     NativeJSContext js;
7:     if (NS_FAILED(js.error))
8:         return js.error;
9:
10:    MakeContextCurrent();
11:
12:    GLint len = 0;
13:    gl->fGetProgramiv(progname, LOCAL_GL_ACTIVE_ATTRIBUTE_MAX_LENGTH, &len);
14:    if (len == 0) {
15:        *retval = nsnull;
16:        return NS_OK;
17:    }
18:
19:    nsAutoArrayPtr<char> name(new char[len]);
20:    PRInt32 attrsize = 0;
21:    PRUint32 attrtype = 0;
22:
23:    gl->fGetActiveAttrib(progname, index, len, &len, (GLint*) &attrsize, (WebGLuint*) &attrtype, name);
24:    if (attrsize == 0 || attrtype == 0) {
25:        *retval = nsnull;
26:        return NS_OK;
27:    }
```

Figure 1: Buggy copy of the clone

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 up-to 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 doesnt 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 changing of 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.

```
1: // File: mozilla\content\canvas\src\webglcontextgl.cpp 1146:14 -> 1172:4
2:
3:   if (!GetGLName<WebGLProgram>("getActiveUniform: program", pobj, &progname))
4:       return NS_OK;
5:
6:    NativeJSContext js;
7:    if (NS_FAILED(js.error))
8:        return js.error;
9:
10:   MakeContextCurrent();
11:
12:   GLint len = 0;
13:   gl->fGetProgramiv(progname, LOCAL_GL_ACTIVE_UNIFORM_MAX_LENGTH, &len);
14:   if (len == 0) {
15:       *retval = nsnull;
16:       return NS_OK;
17:   }
18:
19:   nsAutoArrayPtr<char> name(new char[len + 3]); // +3 because we might have to append "[0]", see below
20:   PRInt32 attrsize = 0;
21:   PRUint32 attrtype = 0;
22:
23:   gl->fGetActiveUniform(progname, index, len, &len, (GLint*) &attrsize, (WebGLenum*) &attrtype, name);
24:   if (len == 0 || attrsize == 0 || attrtype == 0) {
25:       *retval = nsnull;
26:       return NS_OK;
27:   }
```

Figure 2: Fixed copy of the clone

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 code.

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] 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 of their work. The target of this machine learning model is provide a generic solution for detecting this particular type of bug without any manual training and exhibit some significant improvement in accuracy over other existing works.

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 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 hardly any 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. Each data item has only one or two feature related code semantic and most likely conditionally independent of each other. We have tried with Naive Bayes, SVM, and Neural Network to see the effectiveness in detecting

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)

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

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 100%, is probably most desirable for any software project. But this one suffers from low accuracy on detecting False cases.

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%

Table 2: Effectiveness on Reducing False Cases

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

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%

Table 3: Effectiveness on Selective Classification

This dataset selectively considers only three 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 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

4 Future Work

5 Conclusion

References

[Marinescu and Dechter, 2005] Radu Marinescu and Rina Dechter *AND/OR branch-and-bound for graphical models*. International Joint Conference on Artificial Intelligence (IJCAI), 2005.

[Sang and Kautz, 2007] Tian Sang, Paul Beame, and Henry Kautz, *A Dynamic Approach to MPE and Weighted MAX-SAT*. Proceedings of the 20th International Joint Conference on Artificial Intelligence (IJCAI), 2007.

[Gogate and Domingos, 2010] Vibhav Gogate and Pedro Domingos, *Formula-Based Probabilistic Inference*. 26th Conference on Uncertainty in Artificial Intelligence (UAI), 2010.