2011 International Conference on Modeling, Simulation and Control
IPCSIT vol.10 (2011) © (2011) IACSIT Press, Singapore

Fault Prediction Using the Similarity Weights Method

Kidsana Jindarak and Usa Sammapun

Department of Computer Science, Faculty of Science, Kasetsart University,
Bangkok, Thailand

Abstract. Software maintenance is one of difficult activities in the development process and could force developers to put in significant time and effort to find and debug software defects. In this paper, we propose a new and simple method for automatically predicting fault in unlabelled code. Our technique predicts faults that occur in the software by using local alignment to find significant weight and applying SVM algorithm to classify code to pinpoint which pieces of code is buggy. As an experiment, we applied our technique to an open source software project, and preliminary results show that our technique is possible and leads to software quality and fault prediction.

Keywords: fault prediction, bug prediction, source code similarity.

1. Introduction

In a software development process, maintenance can be a source of high costs; more than 90% of total costs in a software project are spent for maintenance [12]. Many programmers or developers put significant effort into finding and debugging software defects. However, it is difficult to find them and thus should not be done manually. Various techniques have been proposed to automatically generate test cases and therefore could alleviate the effort of finding defects. Other techniques gear towards static analysis and instead involve textual analysis or feature extraction of source code to effectively predict various aspects of software qualities [2, 3, 4, 5, 7]. The analysis result should allow developers to identify software defects much faster and therefore decrease maintenance costs.

Our work employs the static analysis technique by combining the similarity-based weighting approach and the SVM classification algorithm to predict software faults. The local alignment, based on dynamic programming, is one of the similarity-based weighting approaches used to find similarity in protein and similarity in text. The SVM (Support Vector Machines) algorithm is a classification technique that needs supervised learning from a training data set to identify features of different classes. The identified features are then used for prediction.

Our technique first identifies pieces of code that could possibly be defects in software by analyzing the software source code in a software configuration management (SCM) system, which tracks and controls changes of the software. The identification is done by first locating change or bug reports from the evolution history that contain information on reported bugs and their subsequent fix. The removed or modified code associated with the bug fix is considered buggy code because it should have been removed or modified in order to be replaced with correct code [3].

Once the buggy code is found, our technique then applies a similarity-based weighting approach using adaptive local alignment [1, 2] to compute similarity scores between the buggy code and other pieces of code. These similarity scores are arranged into a matrix and used as features in the SVM algorithm. The data set computed by local alignment is divided into a training data set and a test data set. The training data set will supervise the SVM algorithm to learn which features within the feature matrix contribute to buggy code. The SVM algorithm thus uses the identified feature to predict which pieces of code in the test data set is buggy code.

2. Related Work

This research is closely related to two research works. The first research entails work related to local alignment while the second research involves bug prediction. In the first research, Jeong-Hoon Ji et al. [1] use local alignment to detect similarity in source code. The local alignment employs the Smith-Waterman algorithm [8] to compute similarity scores. In general, the similarity score for each keyword is +1 for matches, -1 for mismatches, and -2 for matching a keyword with a gap. In addition, the frequencies for keywords are also considered where low frequencies give high scores and high frequencies give low scores.

In the related research on the bug prediction, Kim et al. [3] apply the SVM algorithm to predict bug by classifying changes in software. The same files from two different versions are compared to find regions that are different. Kim et al. call such a region a "hunk." If this hunk is involved in the bug fix and is deleted or modified, it is considered buggy code. The features from these changes such as metadata, source code, or change log message are used for classification in the SVM algorithm.

Our technique adopts the two approaches to be used together where the similarity score matrix resulted from the local alignment is used as feature extraction for the SVM algorithms instead of metadata or other information used in the work by Kim et al. [3].

There are also other related researches that extract feature data from the source code for other prediction approaches such as software metrics for fault prediction [4, 7] and refactoring histories for refactor prediction [5].

3. Methodology

This section illustrates each step of our technique where Section 3.1 describes how source code is analyzed and identified as buggy, Section 3.2 explains how source code is retrieved to be compared with buggy source code, Section 3.3 shows how local alignment and SVM algorithm are used to predict software faults, Section 3.4 describes our experiments, and Section 3.5 provides experimentation results.

3.1. Identifying Buggy code

In order to identify which pieces of code could be defects, source code and its revision information in a software configuration management system is analyzed. The revision information contains summary details about how the code is changed. Our technique looks for terms such as bug, fix, or error in the summary details and assumes that these changes are done to fix bugs.

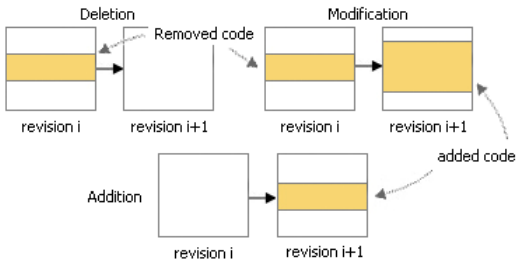

Fig. 1: Types of code changes; removed code and added code.

There are three types of code changes: deletion, modification and addition, as shown in Figure 1. We considered deleted code as defects since it is removed in response to associated bug, fix, or error terms in the summary details. Modified code can be thought of as code removed in the previous version and code added in the newer version, and thus we consider the removed code in the modification type as defects as well. As for code addition, we consider it to be feature enhancement [6] and will not be included in our analysis. In this paper, we use the term "hunk" in the same manner as in Kim et al.'s work [3], which defines a "hunk" as a region of code that is different in two revisions.

Lines 606-621 Link Here

```
606         }                                                   606         }
607                                                             607
608         // declaring type                                   608         // declaring type
            char[] qualifiedPattern = qualifiedPattern                      if (this.pattern.declaringSimpleName == null &&
609 (this.pattern.declaringSimpleName,                         609 this.pattern.declaringQualification == null) return
    this.pattern.declaringQualification);                           methodLevel; // since any declaring class will do
            if (qualifiedPattern == null) return
610 methodLevel; // since any declaring class will do
611                                                             610
            boolean subType = !method.isStatic() && !                       boolean subType = !method.isStatic() && !
612 method.isPrivate();                                         611 method.isPrivate();
            if (subType &&                                                  if (subType &&
    this.pattern.declaringQualification != null &&              612 this.pattern.declaringQualification != null &&
613 method.declaringClass != null &&                                method.declaringClass != null &&
```

Fig. 2: Eclipse JDT project, bug#324189, Lines 606-621.

Consider the defect report in Figure 2 from the Eclipse JDT project with bug #324189. It shows comparison computed by a *diff* tool between two revisions. The left side is a previous source code revision while the right side is the newer one. By searching for keywords with terms such as *bug*, *fix*, or *error*, we record source code from the left side, which had been removed, into the database by using the extracting system described in Section 3.2. We also include package name and file name to identify the location of the removed source code.

3.2. Information Extraction

Code change management could be used by researchers to study approaches for assisting in the maintenance. They can be extracted from their software configuration management (SCM) system to be training sets and test sets for machine learning. The bug fix changes for each file are identified by examining keywords such as *bug*, *fix*, *error* in SCM summary details. We then extract removed code, package name and file name from SCM repository as described in Section 3.1 by using the extract system shown in Figure 3. Note that one source code file can result in various hunks of code.

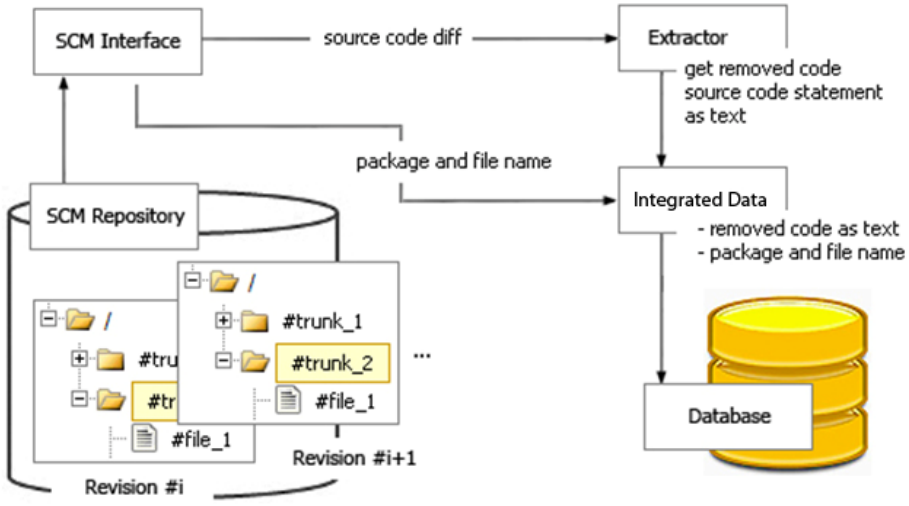

Fig. 3: SCM Extraction system

In the database, we store line numbers of code in order to help locating the code. These line numbers are used when comparing pieces of code during fault prediction and correctness evaluation of the prediction result.

3.3. Similarity Searching and Maximum Similarity Weight

We present our approach to find similarities between source code and buggy code by using local alignment technique to compute similarity scores. In this paper, we will use all of source code syntax instead of just keywords in all source code files to be compared and analyzed. To clean up source code, all spaces between two words are reduced into one single space. All comments are also removed since we only consider the source code itself. After the data cleanup, we calculate the frequency f_i of the syntax k_i in a hunk of code and the frequency f_j of the syntax k_j in buggy code. f_i denotes the number of occurrences of k_i in a hunk of code divided by the number of occurrences in both hunk of code and buggy code. f_j is calculated in a

similar manner as f_i. The frequency values are then used by local alignment algorithm based on dynamic programming to calculate the similarity score S_{ij}, which is defined as follows:

If $k_i = k_j$ then $S_{ij} = -\alpha * \log_2 (f_i * f_j)$ (1)

If $k_i \neq k_j$ then $S_{ij} = \beta * \log_2 (f_i * f_j)$ (2)

If k_i or k_j is a gap then $S_{ij} = 4\beta * \log_2 (f_{i \text{ or } j})$ (3)

The α and β parameters, where α+ β = 1, can be used to tune or adjust the significance of matches, mismatches, and matching with a gap for each project. If the score for matching the syntax is considered more important and should contribute to higher similarity weights, then we will adjust the α value to be higher than the β value. On the other hand, if the score for mismatching the syntax is considered more significant, we will adjust the β value to be higher than the α value. As described in Section 2, the equations (1), (2) and (3) are variants of the weight equations used in the works by Jeong-Hoon Ji et al. [1] and in the Smith-Waterman algorithm [8]. In our variant, we use the gaps to split syntax of source code and consider a space not significantly different from a gap.

The next step in our technique is to prepare the similarity weight data as feature extraction in the SVM algorithm. Each buggy code is compared to each hunk of code to calculate similarity weight. The weight returned by the local alignment will return the highest weight possible for the buggy code and the hunk of code. We represent the buggy code y in a file x as b_{xy}, and the hunk of code m in a file n as h_{mn}. In Figure 4(a), similarity weight between a buggy code b_{xy} and a hunk of code h_{mn} is computed as $W_{mn,xy}$. The value is then inputted into the matrix in Figure 4(b) with all other weights for all buggy code and hunk of code pairs.

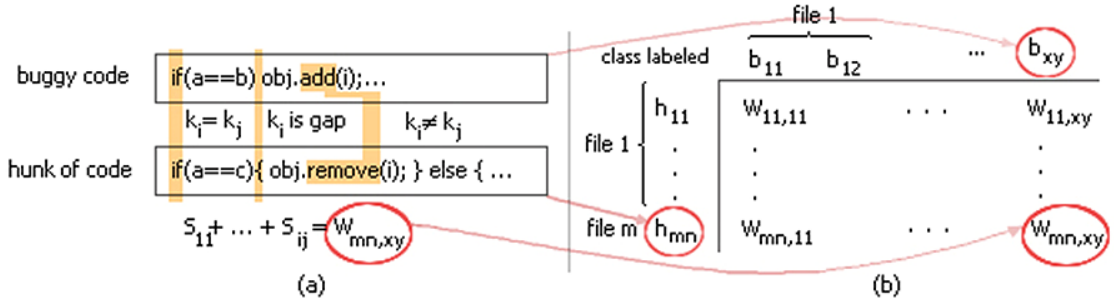

Fig. 4: a: local alignment searching, b: data preparation

To train the data using the SVM algorithm, we need to label hunks of code as *buggy code* and *unknown code*. The code is considered buggy when that hunk of code has overlapping texts with actual buggy code. Unknown code is the code that we do not know exactly whether it is the buggy codes.

The similarity weight matrix shown in Figure 4(b) is already in the general format used in the SVM algorithm [10, 11]. The format is shown as follows.

(class) (feature index)$_1$: (feature value)$_1$ (feature index)$_2$: (feature value)$_2$ ...

The reason why we choose the SVM as our classification algorithm is that SVM has a good behavior when the classification has to deal with a large number of features (more than 10000 features) [9] and our training data set does have a very large number of features.

3.4. Experiment

As our data preparation, we reduced multiple spaces between words into one single space, chopped off all new lines, and removed all comments so that the similarity weight are computed with source code only. We have followed all the steps as shown in Figure 5.

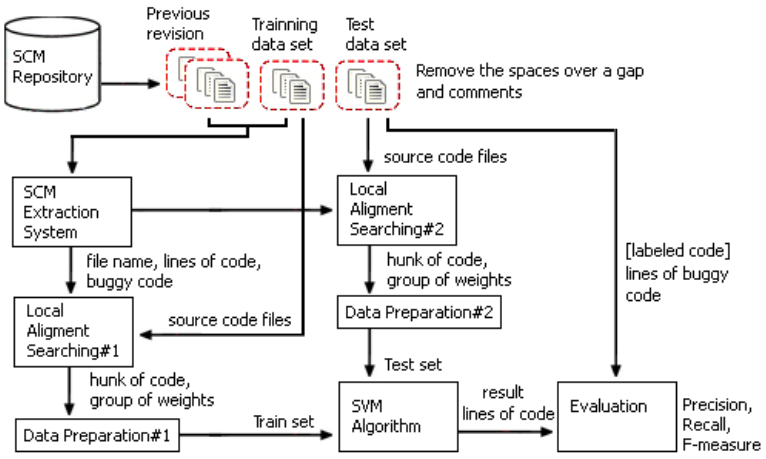

Fig. 5: Overview of method.

To conduct experiments, we prepared sample data of 50 pieces of buggy code as features from the project of the Eclipse's JDT Core version 3.5–3.6. Even though some of these versions were not officially released, these versions had resolved some defects and thus can be used to extract labeled buggy code. 100 hunks of code extracted from five source code files in these revisions are used as a training data set. We also extract hunks of code from other five different files as our test data set. We then use our technique to predict faults from the test data set. The preliminary result of the technique classified 36 hunks of test data codes as buggy codes (correct: 19) and 59 hunks as unknown codes (correct: 43).

3.5. Evaluation of Result

The data sets are divided into two sets: a training data set from history revision and a test data set to be predicted. We evaluate our method by reviewing the test data set and identifying actual defects. If there exists code overlaps between actual defect and the predicted buggy code, then we will consider the result a *correct buggy code prediction*. On the other hand, if the predicted buggy code does not overlap with actual defect at all, we will regard the result as a wrong prediction.

```
/* check first segment */
char patternChar = 0;
while ((iPattern < patternEnd)
        && (patternChar = pattern[iPattern]) != '*') {
        if (iName == nameEnd)
                return false;
        if (patternChar
                != (isCaseSensitive
                        ? name[iName]
```

Buggy code

Correct area

Buggy code prediction

Fig. 6: Buggy code statement and prediction.

In this paper, we evaluate our approach using accuracy, recall, precision, and F-value measures because they are widely used for evaluating classification [3, 5, 7]. However, in our context, accuracy is not a good performance metric because very high accuracy can be achieved by always classifying code as unknown code (false negative class). The recall and precision can be computed using following equations:

Recall = number of *correct buggy code prediction* / total number of buggy code in example. (4)

Precision = number of *correct buggy code prediction* / total number of buggy code prediction. (5)

F-value is a composite measure of buggy code recall and precision:

F-value = 2 * Precision * Recall / (Precision + Recall). (6)

As the preliminary result, our technique can correctly predict 19 the buggy codes, which results in the recall measure at (19/35) 0.54 calculated using equation (4) and the precision value at (19/36) 0.53 calculated using equation (5). That leads to the F-value of 0.535.

4. Conclusion and Future Work

Finding software bugs is a challenging maintenance task. A methodology described by this paper aims to predict software faults in order to lower the cost and simplify software maintenance. The technique used in

this research first determines labeled defects, then find the similarity weights between labeled defect and source code in the training data set, and finally use the resulting weights as feature extraction in SVM algorithm to make a prediction. However, since the prediction uses only similarity weights, it may lead to biased results.

In the future, we will do more experiments on data from other open source software projects to study the effectiveness of the method more thoroughly. We will also apply a method for data cleaning in order to eliminate noisy data. In addition, to better train data, cluster algorithms can be used to improve the data preparation. In the aspect of bug prediction, other prediction techniques will be used to compare the results in order to find the algorithm with the best result.

5. Acknowledgements

The authors thank Kitti Koonsanit for his guidance and insight on writing for this paper. This work was supported by budget for the oversea academic conference from Faculty of Science, Kasetsart University and the Graduate School, Kasetsart University.

6. References

[1] J.-H. Ji, S.-H. Park, G. Woo, and H.-G. Cho, "Source Code Similarity Detection Using Adaptive Local Alignment of Keywords," *IEEE Comput. Society*, *Proc. Eighth International Conf.*, 2007, pp. 179-180.

[2] Zachary P. Fry, "Fault Localization Using Textual Similarities," M.S. thesis, Dept. Comput. Sci., Virginia Univ., Charlottesville, Virginia, 2010.

[3] S. Kim, E. James, and Y. Zhang, "Classifying Software Changes: Clean or Buggy?," *IEEE Trans. Softw. Eng.*, vol. 34, March/April 2008, pp. 181-196.

[4] A. Mahaweerawat, P. Sophatsathit, and C. Lursinsap, "Software Fault Prediction Using Fuzz Clustering and Radial-Basis Function Network," *Proc. Intelligent Technologies Conf.*, December 2002, pp. 304-313,.

[5] J. Ratzinger, T. Sigmund, P. Vorburger, and H. Gall, "Mining Software Evolution to Predict Refactoring," in *Symp. Empirical Software Engineering and Measurement*, September 2007, pp.354-363.

[6] A. E. Hassan, "Automated Classification of Change Messages in Open Source Projects," *Proc. ACM*, *Symp. Applied computing*, 2008, pp.837-841.

[7] H. Zung, "An Investigation of the Relationships between Lines of Code and Defects," *IEEE International Conf.*, *Symp. on Software Maintenance*, *ICSM*, October 2009, pp. 274-283.

[8] T.F. Smith and M.S. Waterman, "Identification of common molecular subsequences," Journal of Molecular Biology, Mar 1981, pp. 147-195.

[9] T. Joachims, "Text Categorization with Support Vector Machine: Learning with Many Relevant Feature," Dept., Informatics, Dortmund Univ., Dortmund, Germany, 1998.

[10] C.-C. Chang, C.-J. Lin (2011, May20). LIBSVM - A Library for Support Vector Machines [Online]. Available: http://www.csie.ntu.edu.tw/~cjlin/libsvm/

[11] T. Joachims (2008, August 14). SVM-HMM Sequence Tagging with Structural Support Vector Machines [Online]. Available: http://www.cs.cornell.edu/People/tj/svm_light/svm_hmm.html

[12] J. Koskinen (2010, September 10). Software Maintenance Costs [Online]. Available: http://users.jyu.fi/~koskinen/smcosts.htm