Optimizing Test Case Execution Schedule using Classifiers

Xiang Chen, Zhaofei Tan, Jian Xia, Pengfei He
School of Computer Science and Technology, Nantong University, Nantong 226019, China
Email: xchencs@ntu.edu.cn

Abstract— **As software evolves, test suite continually grows larger. However running all the test cases in the test suite is prohibitive in most cases. To reduce the cost of regression testing, we can optimize test case execution schedule to maximize the early fault detection rate of the original test suite. Different from previous research, we use classification algorithms to guide the schedule process based on code change information and running result analysis. In particular, we firstly train a classifier for each test case using both the code change information and the running result in previous versions. Then we secondly use the trained classifier to estimate the fault detection probability of the test case in a new version. Finally we generate a test case execution schedule report based on the fault detection probability of all the test cases. To verify the effectiveness of our approach, we performed an empirical study on Siemens Suite, which includes 7 real programs written by C programming language, and chose some typical classification algorithms, such as decision tree classifier, Bayes classifier, or nearest neighbor classifier. Based on the final result, we find that in most cases, our approach can outperform a random approach and then further provide a guideline for achieving cost-effective test case execution schedule when using our approach.**

Index Terms— **software testing, regression testing, test case execution schedule, classification algorithm, empirical study**

I. Introduction

During software development and maintenance, software continually evolves due to fault removal, function modification, or performance improvement. These software evolution behaviors will inevitably cause code changes. Regression testing can be used to guarantee the correctness of these code changes and avoid their side effect to other program modules. Statistical data from some enterprises shows that regressing testing often consumes up to 80% of the total software testing budget and 50% of the software maintenance cost [1]. Test suite maintenance is a core issue in regression testing, however rerunning all the existing test cases is infeasible. For example, Rothermel et al. found that running all the test cases of a software in a cooperation enterprise needs about 7 weeks [2]. A practical solution is to optimize test case execution schedule to maximize the early fault detection rate of the original test suite. This solution is especially useful when the testing budget is limited.

Nowadays different approaches have been proposed to optimize test case execution schedule by prioritizing test cases and they mainly use program entity coverage ability to guide test case execution schedule [2]–[8]. The granularity of program entity can be set as statement, method, or branch. Mirarab and Tahvildari proposed a prioritization approach which is based on bayesian networks [9]. And they further enhanced their approach by incorporating the feedback mechanism [10]. But in their approach, they just assumed that the changes to program entities (such as function) would introduce faults only in the same program entities, but in reality, the changes of program entities would introduce faults to other program entities with data or control dependency. In addition, to construct the training data, they need to consider source code changes, software fault-proneness, and test coverage data. Finally, they used a Bayesian Networks as their classifier which is a heavyweight classifier. To solve this issue, we propose a different approach to optimize test case execution schedule using some lightweight classifiers.

During the process of the software development and maintenance, we can gather the code change information and corresponding introduced faults. From these data, we find that in a specific program module, code changes are prone to introduce new faults. Therefore we can conjecture that there exists some specific relationship between the code change and the fault detection probability, and we want to use classification algorithms to mine these relationship. We firstly construct training data by gathering code change information and running result of each test case in previous versions. We secondly train a classifier for each test case using a specific classification algorithm. We thirdly use the trained classifier to estimate the fault detection probability of this test case in the new version. Finally we prioritize these test cases according to corresponding fault detection probability.

To verify the effectiveness of our approach, we designed and performed an experimental study. In particular, we chose some classical classification algorithms and used Siemens Suite as our benchmark. Final result shows that in most cases, our approach has advantage over a random approach. Among different classifiers, Bayes classifier performs best, while decision tree classifier performs worst. We also find that the effectiveness of

Manuscript received October 10, 2013; revised November 5, 2013; accepted November 14, 2013. © 2005 IEEE.

This work was supported in part by The NSFC Project under Grant No. 61202006, the Nantong Application Research Plan under Grant No. BK2012023, the University Natural Science Research Project of Jiangsu Province under Grant No. 12KJB520014, the Innovation Training Project for College Students under Grant No. 201310304087X and 2013078

doi:10.4304/jsw.9.10.2514-2521

our approach is mainly related to the number of versions or LOC/method. These findings can further guide the use of our approach for achieving cost-effective test case execution schedule.

The main contribution of this paper can be highlighted as follows:

- To the best of our knowledge, we firstly propose a classifier based test case execution schedule approach.
- We designed and performed an empirical study to verify the effectiveness of our proposed approach.

The rest of this paper is organized as follows: Section II provides the background of test case execution schedule and introduces some typical classification algorithms. Section III presents details of our classifier based approach. Section IV presents the experimental study including experimental setup, data analysis, and threats to validity. Section V summarizes the related work and highlights the contribution of our work. Section VI draws a conclude and discusses several potential future work.

II. Background

In this section, we briefly introduce the preliminary of test case execution schedule. The formal description of this issue was given by Rothermel et al. [2] as follows:

Given: a test suite T, the set of permutations of T PT, a function f from PT to a real number.

Problem: Find $T' \in PT$ such that $(\forall T'')(T'' \in PT)(T'' \neq T')[f(T') \geq f(T'')]$.

Different from the description of Rothermel et al., Kim and Porter summarized this process into four steps from probability point of view [11]. These steps are: (1) applying a regression test selection technique to a test suite T, yielding T'. (2) calculating a selection probability for each test case in T'. (3) executing the test case from T' with highest probability which is calculated in step 2 one after another. (4) repeating step 3 until testing resource is exhausted. In this paper, our research work is based on the description of Kim and Porter.

To evaluate the performance of our proposed approach, we need a suitable metric. In previous research, Rothermel et al. proposed a metric APFD to measure the weighted average of the percentage of faults detected during the execution of the test suite [2]. However, this metric involves fault numbers that each test case can reveal. Since it is hard to obtain such information when performing black box testing, therefore in this paper, we choose M_1 [12] as our metric to measure the effectiveness of different approaches. M_1 can be used as a measure to illustrate how rapidly an prioritized test cases can detect faults in black box testing. The value of this metric can be computed as follows:

$$M_1 = \frac{\frac{1}{2} \times \sum_{i=1}^{m}[(2 \times m - 2 \times i + 1) \times f_i]}{m \times \sum_{i=1}^{m} f_i} \quad (1)$$

Here m is the size of the test suite and f_i represents whether the i-th test case can detect the fault, if not, the value is 0, otherwise the value is 1.

A. Classification Algorithms

In this subsection, we will illustrate some representative classification algorithms. The classification based learning contains two stages: the first stage is to train a classifier according to the training data, and the second stage is to use the trained classifier to compute the probability of present attribute set to one of the predefined class label. In our research, we choose three representative classification algorithms, they are Bayes classifier, nearest neighbor classifier, and decision tree classifier.

Decision tree is a hierarchical structure, which consists of nodes and directed edges. The tree has three types of nodes: a root node, internal nodes, and terminal nodes. Decision tree has the following metrics: decision tree induction is a nonparametric approach for building classification model. Constructing decision trees is computationally inexpensive. Decision trees are relative easy to interpret. Decision trees are robust to the presence of noise.

Bayes classifier is a simple probabilistic classifier based on Bayes Theorem. It is used to find the relationship between the attributes set and class label. It uses Bayes formula to predict some particular attribute sets which class they belong to. Compared to other classifiers, Bayes classifier can use less time to training and predicating. It is also robust even there exists isolated noise points. There are two main variant techniques. One is AODE (Averaged One-Dependence Estimators) and it was developed to address the attribute-independence problem of the Naive Bayes classifier, it can reduce the requirement of the independence between the attributes. The other is HNB (Hidden Naive Bayes), in this technique, a hidden parent is created for each attribute which can combine the influences from all other attributes. This technique solves the assumption of the independence between attributes and combines the merits of Bayes and Bayes network, therefore it has been succesfully used in many real world applications.

Nearest neighbor classifier is a lazy classifier, it do not use the training data to train the classifier, but to predict directly by using training data. Therefore it can be easily interfered by noises. This technique projects each instance into a point in d-dimension space and determines which class it belongs to by voting of the nearest neighbors. KNN (K-nearest neighbor) is one of the classical nearest neighbor classifiers, it uses the k-nearest neighbors to determine the class which the point belongs to.

III. Our Approach

In this section, we will introduce the framework of our approach used for test case execution schedule in detail.

A. Preliminaries

Before introducing our approach, we formalize some concepts in this subsection.

Let $T = \{t_1, t_2, \cdots, t_m\}$ be the test suite, $V_0 = \{m_1, m_2, \cdots, m_l\}$ be the original version, where m_k represent k-th program entity. In our research, we treat each

function as a program entity. Let $V = \{V_1, V_2, \cdots, V_n\}$ be the version set and each version was performed a code change operation on the original version V_0.

Based on code change analysis and test case running result analysis, we can construct two matrices: code change matrix (CCM) and running result matrix (RRM). Code change matrix can denote the change information between the base version V_0 and the i-th version V_i. In this binary matrix, columns correspond to program entities and rows correspond to versions in V. If the j-th program module is modified in Version V_i, the value of $c_{i,j}$ is 1, otherwise the value of $c_{i,j}$ is 0. Typical code change operators can be summarized into three categories: code addition operators, code modification operators, and code deletion operators.

$$CCM_{n\times l} = \begin{array}{c} \\ V_1 \\ V_2 \\ \cdots \\ V_n \end{array} \begin{array}{c} \begin{array}{cccc} m_1 & m_2 & \cdots & m_l \end{array} \\ \begin{pmatrix} c_{1,1} & c_{1,2} & \cdots & c_{1,l} \\ c_{2,1} & c_{2,2} & \cdots & c_{2,l} \\ \cdots & \cdots & \cdots & \cdots \\ c_{n,1} & c_{n,2} & \cdots & c_{n,l} \end{pmatrix} \end{array}$$

Running result matrix can denote the running result of each test case in different versions in V. In this binary matrix, columns correspond to test cases and rows correspond to versions in V. If the j-th test case is passed in Version V_i, the value of $r_{i,j}$ is 1. Otherwise the j-th test case is failed in Version V_i, the value of $r_{i,j}$ is 0.

$$RRM_{n\times m} = \begin{array}{c} \\ V_1 \\ V_2 \\ \cdots \\ V_n \end{array} \begin{array}{c} \begin{array}{cccc} t_1 & t_2 & \cdots & t_m \end{array} \\ \begin{pmatrix} r_{1,1} & r_{1,2} & \cdots & r_{1,l} \\ r_{2,1} & r_{2,2} & \cdots & r_{2,l} \\ \cdots & \cdots & \cdots & \cdots \\ r_{n,1} & r_{n,2} & \cdots & r_{n,l} \end{pmatrix} \end{array}$$

Based on these two matrices, for the test case t_i, we can construct a collection of records which can be used as training data. The attribute set of the record can be gathered from matrix CCM, and the class label (i.e., target attribute) can be gathered from matrix RRM. The final records can be shown as follows:

$$\begin{array}{c} \\ 1 \\ 2 \\ \cdots \\ n \end{array} \begin{array}{c} \begin{array}{ccccc} m_1 & m_2 & \cdots & m_l & class \end{array} \\ \begin{pmatrix} c_{1,1} & c_{1,2} & \cdots & c_{1,l} & r_{1,i} \\ c_{2,1} & c_{2,2} & \cdots & c_{2,l} & r_{2,i} \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ c_{n,1} & c_{n,2} & \cdots & c_{n,l} & r_{n,i} \end{pmatrix} \end{array}$$

When considering a new version V_{new}, we can construct a vector $(c_{new,1}, c_{new,2}, \cdots, c_{new,l})$ by analyzing the difference between V_{new} and V. Then we can use this vector and the trained classifier to estimate the fault detection probability of t_i in V_{new}.

B. The Framework of Our Approach

Before introducing our approach, we mainly make the following two assumptions.

Assumption 1: After performing a code change operation, developers often inevitably introduce new faults. Therefore we should perform regression testing to guarantee that no fault is introduced in the new version.

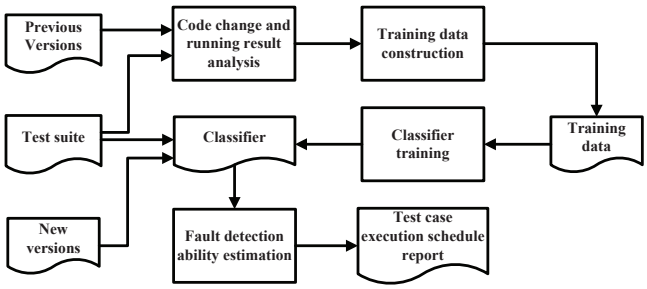


Figure 1. The Framework of Our Approach

Assumption 2: The code change operation is not complex. When the software becomes stable, in most cases, developers often change only one program module. Therefore the new version is almost the same as the previous version.

We use Figure 1 to show the framework of our approach. From this figure, we can find that our approach mainly consists of three phases.

Phase 1. Training data construction: In this phase, we will collect the versions' change information and the running results of each test case in previous versions. Based on these data, we can construct training data of each test case for the Phase 2.

Phase 2. Classifier training: In this phase, we can train a classifier for each test case based on the training data, which is constructed in Phase 1. In our empirical studies, we will use different classification algorithms.

Phase 3. Test case execution schedule report generation: In this phase, for each test case, we will use both the corresponding classifier and the change information in the new version to estimate the fault detection probability of each test case. By using the fault detection probability, we can prioritize these test cases in the descending order and finally return the test case execution schedule report.

IV. Empirical Study

To investigate the effectiveness of our approach, we conducted an empirical study and wanted to answer the following three research questions:

RQ1: How about the effectiveness of our novel classification algorithm based approach when comparing a random approach?

RQ2: Which classification algorithm is more suitable for our approach in the empirical study?

RQ3: Which characteristic of empirical subject is more suitable for our approach in the empirical study?

A. Independent Variables and Dependent Variables

In the design of empirical studies, we usually control or change certain factors to investigate the relationships between these factors and experimental results. These factors are called independent variables. In this paper, we only consider one independent variable (i.e., classification

TABLE I.
THE SUMMARY INFORMATION OF EXPERIMENTAL SUBJECTS

Subject Name	LOC	#Methods	#Versions	#Tests
printtokens	726	18	7	4130
printtokens2	570	19	10	4110
schedule	412	18	9	2650
schedule2	374	16	10	2710
replace	564	21	32	5542
totinfo	565	7	23	1052
tcas	173	9	41	1608

algorithms in our approach). In empirical studies, we use dependent variables to measure the final experimental results. In this paper, we only consider M_1 metric and the detail is illustrated in Section II.

B. Experimental Subjects, Test Suits, and Faults

We use seven open-source subjects in Table 1, these subjects were developed by Siemens researchers [13] and further developed by Rothermel et al. to make sure that each executable statement, definition-use pair, or branch should be covered at least by 30 test cases. These seven subjects are also called Siemens suite and they are commonly used in empirical studies of software testing [2]–[4], [7], [14], [15]. Table I shows the summary information of these seven subjects, such as lines of code (LOC), the number of methods the each subject has, the number of mutant versions, and the size of test pool. Each subject has a original version and the number of mutant versions ranges from 7 to 41. In the mutant versions, the mutant is seeded by researchers who have rich project development experience. Most of the mutant operators just modify only one line.

In our experimental studies, we randomly generated 100 test suits from the original test case pool, the size of each test suite is set to 100. Then we will apply our approach to each test suite to verify the effectiveness of our approach.

C. Experimental Setup

As show in Figure 1, our framework contains three phases. These phases are training data construction, classifier training, and test case execution schedule report generation in sequence. In this subsection, we will explain the implementation details in these three phases respectively.

Phase 1. Training data construction: In this phase, we mainly perform code change analysis and running result analysis. During code change analysis, we use Adiff tool [1] to help us to construct CCM, which is defined in Section III. Adiff is an automatic differentiation utility and can be used to collect version change information between two different versions. During running result analysis, we run each test case on different versions in the version set V and then construct RRM, which is defined in Section III. When the test case outputs an expected value, we call it a passed test case, else we call it a failed test case.

Phase 2. Classifier training: In this phase, we choose WEKA (Waikato Enironment for Knowledge Analysis) [2] as our classifier training tool. This tool is developed by the University of Waikato in New Zealand and now is widely used for data mining by many researchers. In our empirical studies, we chose five typical classifiers such as HNB, AODE ,IBk, ADTree, and BFTree. The previous two classifiers (i.e., HNB and AODE) are based on Bayes classification, IBk is a classifier based on nearest neighbor classification, and BFTree is a classifier based on decision tree classification.

Phase 3. Test case execution schedule report generation: In this phase, we will use the trained specific classifier to predict the fault detection probability of each test case in the new version. When we get the fault detection probability of all the test cases, we can use a specific sorting algorithm to schedule test case execution order in the descending way. In our study, we use quick sorting algorithm.

D. Data Analysis

In this subsection, we summarize all the data gathering from the empirical study and answer the three research questions.

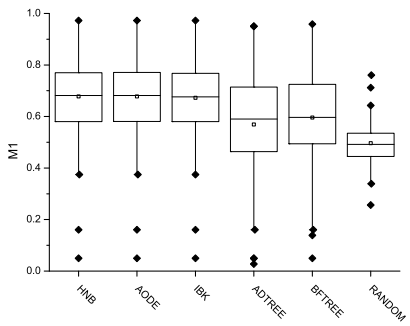
In the empirical study, we use a leave-one-out method to evaluate the performance of our approach. In particular, in i-th run, we use V_i as new version, and $V - V_i$ as the previous versions. This procedure is repeated n times. Therefore when choosing a specific classification algorithm, we can get n different M_1 values for each subject. If version V_i is chosen as the new version, the value of M_1 is denoted as M_1^j. Finally we can compute the mean value to evaluate the performance of our approach and the formula is:

$$\overline{M_1} = \frac{\sum_{j=1}^{n} M_1^j}{n} \tag{2}$$

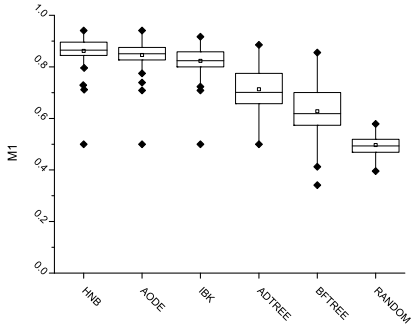
1) Effectiveness of Our Approach: To answer RQ1, we compare our approach with a random approach. When using the random approach, we schedule test case execution order randomly. Since there exist 100 different test suites for each subject, the distribution of $\overline{M_1}$ value is shown in Figure 2. Except for one exceptional case (i.e., use ADTree classifier for schedule2 subject, the $\overline{M_1}$ value of our approach is smaller than the random approach), our approach can perform better than the random approach. From Figure 2, we can find that when using the random approach, the mean value of $\overline{M_1}$ is about 0.5. However, when using our approach, the mean value of $\overline{M_1}$ ranges from 0.5 to 0.9. Especially, when analyzing subjects printtokens2, totinfo, replace, and tcas, the $\overline{M_1}$ value is significantly better than the random approach which can indicate that if using our approach, we can as early as possible to find out more faults.

[1] http://mathforum.org/library/view/70812.html, Accessed in Aug. 2013

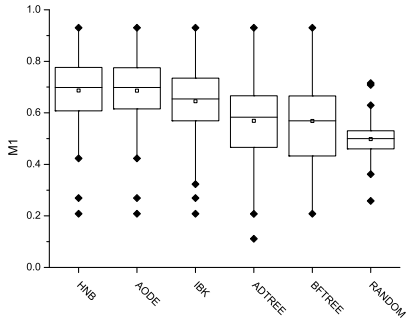
[2] http://www.cs.waikato.ac.nz/ml/weka/, Accessed in Aug. 2013



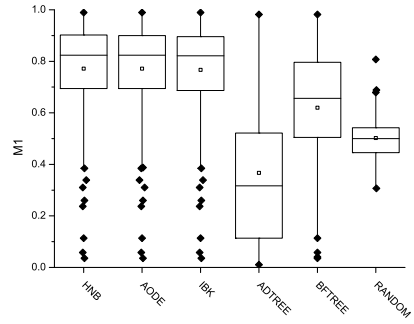
(a) printtok



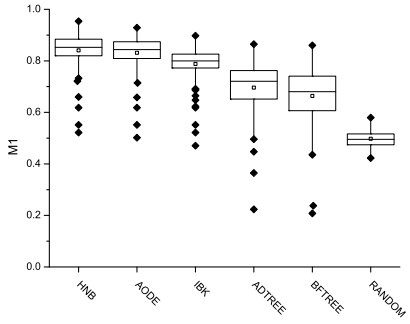
(b) printtok2



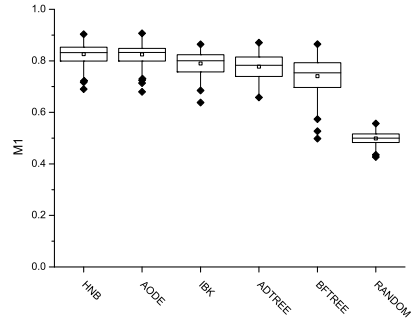
(c) schedule



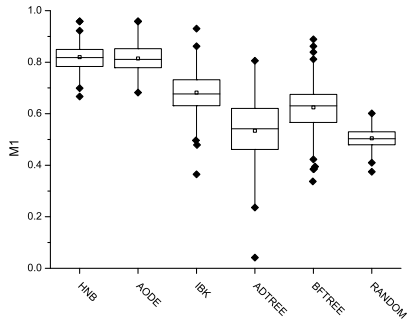
(d) schedule2



(e) replace



(f) totinfo



(g) tcas

Figure 2. The $\overline{M_1}$ distribution for each subject

2) Effectiveness of Different Classification Algorithms: To answer RQ2, we use Table II to record the mean value of $\overline{M_1}$. In Table II, we add "Optimal" column. Since this is a controlled experimental study, we can know the running result of each test case on the new version in advance. Therefore we can get an optimal test case execution order and compute the corresponding M_1 value. The last row shows the mean value of $\overline{M_1}$ for each classification algorithm over all the subjects.

According to the summary data of Table II, we can find that HNB and AODE perform better than other classification algorithms. IBk is the second, and ADTree and BFTree perform worst among these algorithms. If we use HNB and AODE , the $\overline{M_1}$ values can achieve more than 10% higher than other classification algorithms. We can find that the $\overline{M_1}$ is nearly 0.79 if we use HNB and AODE, and is nearly 0.73 if we use IBk, and is nearly 0.60 if we use ADTree and BFTree. In summarization, in most subjects Bayes classifiers are the most suitable for our approach, nearest neighbor classifiers are less suitable for our approach, and decision tree classifiers are worst for our approach.

3) Effectiveness of Different Characteristics of Subjects: After answering RQ2 we can find that the Bayes classifiers maybe the most suitable for our approach than other classifiers. Therefore we will further analyze the effectiveness of different subjects in our approach based on the classification algorithms HNB and AODE. As we can find from Figure 2, our approach can perform better in the subjects, such as replace, schedules2, tcas, and totinfo. Table III shows the relationship between the mean value of $\overline{M_1}$, which is on HNB and AODE, and the attributes of subjects. Here column "LOC/method" (LOCM) denotes the average LOC of each method. From this table, we can find that three subjects (i.e., replace, schedules2, and totinfo) which performs better usually have larger LOCM. The value of LOCM is all more than 25. However, we also note that the LOCM of tcas is the smallest, but our approach can still perform well on this subject. We conjecture the reason is that the mutant number of this subject is larger than other subjects. Since when we use the classification algorithms, we know that the more number of training data, the better classification models we can get. In summarization, from Table III, we can find that the subjects with more mutant number or larger LOCM maybe the most suitable for our approach.

The findings of RQ2 and RQ3 can provide a guideline for achieving cost-effective test case execution schedule when using our proposed approach.

E. Threats to Validity

In this section, we mainly discuss the potential threats to validity of our research.

Threats to external validity are about whether the observed experimental results and conclusion can be generalized to other subjects. One external threat is that the subjects we have used are all written by C programming language. Therefore the conclusion may not be applicable to other subjects written by other programming languages. However these subjects are widely used by other researchers in their empirical studies [2]–[4], [7]. Another external threat is the chosen classification algorithms. There maybe exist other classification algorithms which can perform better then the chosen classification algorithms in our research work. However our chosen classification algorithms is the most classical in traditional data mining textbooks.

Threats to internal validity are mainly concerned with the uncontrolled internal factors that might have influence on the experimental results. The main internal threat is the process of information collection and the correctness of our programs. To avoid these issues, we firstly downloaded all the subjects, mutant versions, and test cases from SIR repository[3]. We secondly examined the results carefully and wrote additional verifying programs to guarantee the correct implementation of our approach.

Threats to construct validity are about whether the metrics used in the experimental study reflect the real-world situation. In previous research, researchers mainly use APFD (average percentage of fault detection) to assess the effectiveness of their proposed approach. However this metric is particular suitable for the general test case prioritization issue. In this issue, testers want to prioritize test cases and hope to be useful on a set of modified versions. While in the specific test case prioritization issue, we hope that the test case prioritization is useful on a specific version and in this paper we mainly focus on this issue and use M_1 as our experimental metric.

V. Related Work

Regression testing is frequently performed to guarantee the correctness of software under test as it continuously evolves. In practice, software tester can use test case repair, test case selection, test suite augmentation, test suite reduction, and test case prioritization to improve the quality of regression test suite [16]–[19]. In this paper, we mainly focus on test case prioritization, this technique aims to schedule test case execution order to improve the early fault detection rate of the original test suite. Except for traditional software testing, the achievement of this issue has also been applied to other specific application domains, such as configurable software [20], [21], GUI [22], Web application [23], [24], Web service [25], and fault localization [26].

The existing research work on traditional software testing can be summarized into three categories: (1) greedy algorithm based, (2) machine learning based, and (3) expert knowledge based. Greedy algorithm based approaches aim to prioritize test cases based on the coverage ability of program entities. Commonly adopted program entity includes statement, branch, method, or MC/DC (Modified Condition/Decision Coverage) [2]–[4], [6], [7], [27]. Based on whether using feedback, the prioritization strategies can be divided into total strategies and additional strategies [2], [3]. Some researchers use machine

[3]http://sir.unl.edu/portal/index.php, Accessed in Aug. 2013

TABLE II.
THE MEAN VALUE OF $\overline{M_1}$ FOR EACH CLASSIFICATION ALGORITHM

Subject	Optimal	HNB	AODE	IBk	ADTree	BFTree	Random
printtokens	0.9916	0.6779	0.6782	0.6728	0.5689	0.5960	0.4967
printtokens2	0.9728	0.8619	0.8459	0.8234	0.7135	0.6278	0.4968
schedule	0.9832	0.6868	0.6864	0.6455	0.5688	0.5686	0.4942
schedule2	0.9442	0.7717	0.7717	0.7668	0.3669	0.6202	0.5022
replace	0.9439	0.8409	0.8311	0.7878	0.6962	0.6637	0.4978
totinfo	0.9598	0.8259	0.8246	0.7819	0.5342	0.6247	0.4988
tcas	0.9762	0.8198	0.8146	0.6819	0.5342	0.6247	0.5045
AVG	0.9674	0.7836	0.7789	0.7372	0.5690	0.6180	0.4987

TABLE III.
THE RELATIONSHIP BETWEEN $\overline{M_1}$ VALUE AND SUBJECTS' ATTRIBUTES

Subject	$\overline{M_1}$	LOC	#Methods	#Versions	LOC/method
printtokens	0.6780	726	18	7	40.33
printtokens2	0.8359	570	19	10	30.00
schedule	0.6866	412	18	9	22.89
schedule2	0.7717	374	16	10	23.38
replace	0.8360	564	21	32	26.86
totinfo	0.8253	565	7	23	80.71
tcas	0.8172	173	9	41	19.12

learning approach to schedule test case execution. For example, Li et al. used hill climbing and genetic algorithm [7], [28]. Mirarab and Tahvildari used Bayesian network [9], [10]. Leon and Podguski used cluster analysis [14]. Carlson further considered code coverage information, code complexity, and previous fault detection information of test cases in their cluster analysis [29]. Researchers also notice that when scheduling test case execution order, we can use expert knowledge to further improve the effectivness. Tonella et al. proposed a case-based ranking approach [30]. Yoo et al. incorporated expert knowledge into cluster analysis [15].

In some software testing scenarios, the testing budget for software is limited to running all the test cases. Researchers named this issue as time-aware test suite prioritization. Kim and Porter firstly research this issue [11]. Then Walcott et al. used genetic algorithm [31] and Zhang et al. used integer linear programming [32] to solve this issue respectively. Do et al. further analyzed how the time constraint affects the existing test case prioritization techniques [33].

Based on the summarization of related work, we find that Mirarab and Tahvildari also used a classifier to assist test case execution schedule, but they (1) constructed training data by considering source code changes, software fault-proneness, and test coverage data, however gathering these data is computational expensive. (2) utilized Bayesian Networks as their classifier which its structure is complex and needs setting more parameter values. Different from their research, we firstly propose a classifier based framework which only uses code change information and running result of each test case in previous versions. These information can be easily obtained in real software testing process. Moreover, we chose different lightweight classification algorithms, such as decision tree classifier, Bayes classifier, and nearest neighbor classifier. Finally we designed and performed an empirical study to verify the effectiveness of our framework and further analyzed the influencing factors in this framework.

VI. CONCLUSION AND FUTURE WORK

In this paper, we proposed a classification based test suite prioritization technique and conducted a set of empirical studies to verify the effectiveness of our approach.

As a preliminary research, there are some issues needed to be solved. In the future work, we firstly want to consider more classification algorithms to augment our proposed framework. Secondly we want to conduct more experimental studies. In particular, we want to adopt more subjects written by other programming languages or more subjects coming from real world development. Last but not the least, we want to incorporate our proposed framework into reality software development process.

REFERENCES

[1] H. K. N. Leung and L. White, "Insights into testing and regression testing global variables," *Journal of Software Maintenance*, vol. 2, no. 4, pp. 209–222, 1990.
[2] G. Rothermel, R. J. Untch, and C. Chu, "Prioritizing test cases for regression testing," *IEEE Transactions on Software Engineering*, vol. 27, no. 10, pp. 929–948, 2001.
[3] S. Elbaum, A. G. Malishevsky, and G. Rothermel, "Test case prioritization: A family of empirical studies," *IEEE Transactions on Software Engineering*, vol. 28, no. 2, pp. 159–182, 2002.
[4] J. A. Jones and M. J. Harrold, "Test-suite reduction and prioritization for modified condition/decision coverage," *IEEE Transactions on Software Engineering*, vol. 29, no. 3, pp. 195–209, 2003.
[5] H. Do, G. Rothermel, and A. Kinneer, "Empirical studies of test case prioritization in a junit testing environment," in *Proceedings of the International Symposium on Software Reliability Engineering*, 2004, pp. 113–124.
[6] H. Mei, D. Hao, L. Zhang, L. Zhang, J. Zhou, and G. Rothermel, "A static approach to prioritizing junit test cases," *IEEE Transactions on Software Engineering*, vol. 38, no. 6, pp. 1258–1275, 2012.

[7] Z. Li, M. Harman, and R. M. Hierons, "Search algorithms for regression test case prioritization," *IEEE Transactions on Software Engineering*, vol. 33, no. 4, pp. 225–237, 2007.
[8] B. Jiang, Z. Zhang, W. K. Chan, and T. H. Tse, "Adaptive random test case prioritization," in *Proceedings of the International Conference on Automated Software Engineering*, 2009, pp. 233–244.
[9] S. Mirarab and L. Tahvildari, "A prioritization approach for software test cases based on bayesian networks," in *Proceedings of the International Conference on Fundamental Approaches to Software Engineering*, 2007, pp. 276–290.
[10] ——, "An empirical study on bayesian network-based approach for test case prioritization," in *Proceedings of the International Conference on Software Testing, Verification, and Validation*, 2008, pp. 278–287.
[11] J.-M. Kim and A. Porter, "A history-based test prioritization technique for regression testing in resource constrained environments," in *Proceedings of the International Conference on Software Engineering*, 2002, pp. 119–129.
[12] B. Qu, C. Nie, B. Xu, and X. Zhang, "Test case prioritization for black box testing," in *Proceedings of the Annual International Computer Software and Applications Conference*, 2007, pp. 465–474.
[13] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand, "Experiments of the effectiveness of dataflow- and controlflow-based test adequacy criteria," in *Proceedings of the International Conference on Software Engineering*, 1994, pp. 191–200.
[14] D. Leon and A. Podgurski, "A comparison of coverage-based and distribution-based techniques for filtering and prioritizing test cases," in *Proceedings of the International Symposium on Software Reliability Engineering*, 2003, pp. 442–453.
[15] S. Yoo, M. Harman, P. Tonella, and A. Susi, "Clustering test cases to achieve effective and scalable prioritisation incorporating expert knowledge," in *Proceedings of the International Symposium on Software Testing and Analysis*, 2009, pp. 201–212.
[16] L. Zhang, X. Chen, Q. Gu, H. Zhao, X. Shi, and D. Chen, "Catesr: Change-aware test suite reduction based on partial coverage of test requirements," in *Proceedings of the International Conference on Software Engineering & Knowledge Engineering*, 2012, pp. 217–224.
[17] X. Sun, B. Li, C. Tao, and Q. Zhang, "Using fca-based change impact analysis for regression testing," in *Proceedings of the International Conference on Software Engineering & Knowledge Engineering*, 2012, pp. 452–457.
[18] X. Chen, L. Zhang, Q. Gu, H. Zhao, Z. Wang, X. Sun, and D. Chen, "A test suite reduction approach based on pairwise interaction of requirements," in *Proceedings of the Symposium on Applied Computing*, 2011, pp. 1390–1397.
[19] X. Zhang, Q. Gu, X. Chen, J. Qi, and D. Chen, "A study of relative redundancy in test-suite reduction while retaining or improving fault-localization effectiveness," in *Proceedings of the Symposium on Applied Computing*, 2010, pp. 2229–2236.
[20] X. Qu, M. B. Cohen, and G. Rothermel, "Configuration-aware regression testing: an empirical study of sampling and prioritization," in *Proceedings of the International Symposium on Software Testing and Analysis*, 2008, pp. 75–86.
[21] H. Srikanth, M. B. Cohen, and X. Qu, "Reducing field failures in system configurable software: Cost-based prioritization," in *Proceedings of the International Symposium on Software Reliability Engineering*, 2009, pp. 61–70.
[22] C.-Y. Huang, J.-R. Chang, and Y.-H. Chang, "Design and analysis of gui test-case prioritization using weight-based methods," *Journal of Systems and Software*, vol. 83, no. 4, pp. 646–659, 2010.
[23] S. Sampath, R. C. Bryce, G. Viswanath, V. Kandimalla, and A. G. Koru, "Prioritizing user-session-based test cases for web applications testing," in *Proceedings of the International Conference on Software Testing, Verification, and Validation*, 2008, pp. 141–150.
[24] R. C. Bryce, S. Sampath, and A. M. Memon, "Developing a single model and test prioritization strategies for event-driven software," *IEEE Transactions on Software Engineering*, vol. 37, no. 1, pp. 48–64, 2011.
[25] L. Mei, W. K. Chan, T. H. Tse, and R. G. Merkel, "Xml-manipulating test case prioritization for xml-manipulating services," *Journal of Systems and Software*, vol. 84, no. 4, pp. 603–619, 2011.
[26] B. Jiang, Z. Zhang, W. K. Chan, T. H. Tse, and T. Y. Chen, "How well does test case prioritization integrate with statistical fault localization?" *Information and Software Technology*, vol. 54, no. 7, pp. 739–758, 2012.
[27] D. Jeffrey and N. Gupta, "Experiments with test case prioritization using relevant slices," *Journal of Systems and Software*, vol. 81, no. 2, pp. 196–221, 2008.
[28] S. Li, N. Bian, Z. Chen, D. You, and Y. He, "A simulation study on some search algorithms for regression test case prioritization," in *Proceedings of the International Conference on Quality Software*, 2010, pp. 72–81.
[29] R. Carlson, H. Do, and A. Denton, "A clustering approach to improving test case prioritization: An industrial case study," in *Proceedings of the International Conference on Software Maintenance*, 2011, pp. 382–391.
[30] P. Tonella, P. Avesani, and A. Susi, "Using the case-based ranking methodology for test case prioritization," in *Proceedings of the International Conference on Software Maintenance*, 2006, pp. 123–133.
[31] K. R. Walcott, M. L. Soffa, G. M. Kapfhammer, and R. S. Roos, "Timeaware test suite prioritization," in *Proceedings of the International Symposium on Software Testing and Analysis*, 2006, pp. 1–12.
[32] L. Zhang, S.-S. Hou, C. Guo, T. Xie, and H. Mei, "Time-aware test-case prioritization using integer linear programming," in *Proceedings of the International Symposium on Software Testing and Analysis*, 2009, pp. 213–224.
[33] H. Do, S. Mirarab, L. Tahvildari, and G. Rothermel, "The effects of time constraints on test case prioritization: A series of controlled experiments," *IEEE Transactions on Software Engineering*, vol. 36, no. 5, pp. 593–617, 2010.

Xiang Chen received his PhD degree in computer software and theory from Nanjing University in 2011. He is an assistant professor at the School of Computer Science and Technology, Nantong University, China. His research interests include regression testing, combinatorial testing, mutation testing, and fault localization.

Zhaofei Tan is an undergraduate at the School of Computer Science and Technology, Nantong University, China. His research interest is regression testing.

Jian Xia is an undergraduate at the School of Computer Science and Technology, Nantong University, China. His research interest is test case generation.

Pengfei He is an undergraduate at the School of Computer Science and Technology, Nantong University, China. His research interest is regression testing.