Is This a Bug or an Obsolete Test?

Dan Hao[1], Tian Lan[1], Hongyu Zhang[2], Chao Guo[1], Lu Zhang[1]

[1]Key Laboratory of High Confidence Software Technologies (Peking University), MoE
Institute of Software, School of Electronics Engineering and Computer Science
Peking University, Beijing, 100871, P. R. China
[2]Tsinghua University, 100084, P. R. China
{haod,lantian12,guochao09,zhanglu}@sei.pku.edu.cn, hongyu@tsinghua.edu.cn

Abstract. In software evolution, developers typically need to identify whether the failure of a test is due to a bug in the source code under test or the obsoleteness of the test code when they execute a test suite. Only after finding the cause of a failure can developers determine whether to fix the bug or repair the obsolete test. Researchers have proposed several techniques to automate test repair. However, test-repair techniques typically assume that test failures are always due to obsolete tests. Thus, such techniques may not be applicable in real world software evolution when developers do not know whether the failure is due to a bug or an obsolete test. To know whether the cause of a test failure lies in the source code under test or in the test code, we view this problem as a classification problem and propose an automatic approach based on machine learning. Specifically, we target Java software using the JUnit testing framework and collect a set of features that may be related to failures of tests. Using this set of features, we adopt the Best-first Decision Tree Learning algorithm to train a classifier with some existing regression test failures as training instances. Then, we use the classifier to classify future failed tests. Furthermore, we evaluated our approach using two Java programs in three scenarios (within the same version, within different versions of a program, and between different programs), and found that our approach can effectively classify the causes of failed tests.

1 Introduction

After software is released to its user, it is still necessary for developers to modify the released software due to enhancement, adaptation or bug[1] fixing [32], which is typically referred to as software evolution. As estimated, 50%-90% of total software development costs [31, 46, 16] are due to software evolution.

In software evolution, developers usually perform regression testing to make sure that their modifications to the software work as expected and do not introduce new faults. Modifications in software evolution include changing functionality, fixing bugs, refactoring, restructuring code, and so on.

In software evolution, some modifications (e.g., changing functionality) may further imply changes of the specifications on program behaviors. In such cases,

[1] Bugs and faults are used interchangeably in this paper.

developers may need to modify the corresponding tests to reflect developers' changing expectation on program behaviors. Thus, when a regression test fails[2], it may just indicate that the specification the test represents is obsolete and the test itself needs repair. If developers do not want to change their specifications of the program in software evolution, a regression test failure may indicate that developers should modify the program without modifying the test. In other words, the tests and the software under test should be developed and maintained synchronously [59].

Specifically, developers for software systems often reuse and adapt existing tests to evolve test suites [40]. As these existing tests are developed for the old version of the software, when some of them cause failures of the new version in regression testing, the failures may not be due to bugs introduced in the modifications [36, 8]. That is to say, some failures are due to bugs in the modified source code under test, but other failures are due to the obsoleteness[3] of some tests. Specifically, an internal study of ABB Corporate Research indicates that around 80% of failures in regression testing are due to bugs in the software under test and the other failures are due to obsolete tests [45].

As testing frameworks like JUnit[4] have been widely used in practice, the tests constructed by developers are also pieces of code. For example, Fig. 1 presents a Java program and its tests. To distinguish the tests and software under test, we denote the source code of the tests and the source code of the software under test as *test code* and *product code* to make our presentation concise. In practice, it is necessary for developers to identify whether such a failure is caused by a bug in the source code of the software under test or an obsolete test. Otherwise, developers would not know how the regression test failures reflect the quality of the software under test. However, as it may be challenging to guess developers' intention as the product code does not reflect developers' intention(i.e., whether or not changing the specifications of the program) explicitly in software evolution, it is not straightforward to know whether the failure of a regression test is due to faulty program changes or obsolete specifications represented by tests. Moreover, as reported in a technical report from Microsoft Research, the test code is often larger than the product code in many projects [50]. As both the test code and the product code are large, it is tedious and difficult for developers to determine the cause of a failure by manually examining the test code and/or the product code. Furthermore, if developers take obsolete tests as bugs in the software, they may submit some false bug reports and thus incur extra burdens for bug-report processing (e.g., triaging [54]).

Furthermore, the knowledge about whether regression test failures are due to bugs or obsolete tests not only can help understand what is going on in the

[2] If the output of a test is as expected (i.e., being as asserted), we call that the test passes; otherwise, we call that the test fails.

[3] Due to the difference between the new version and the old version of software under test, some existing tests for the old version cannot be used to test the new version. We call tests that need modification for the new version as obsolete tests.

[4] http://www.junit.org.

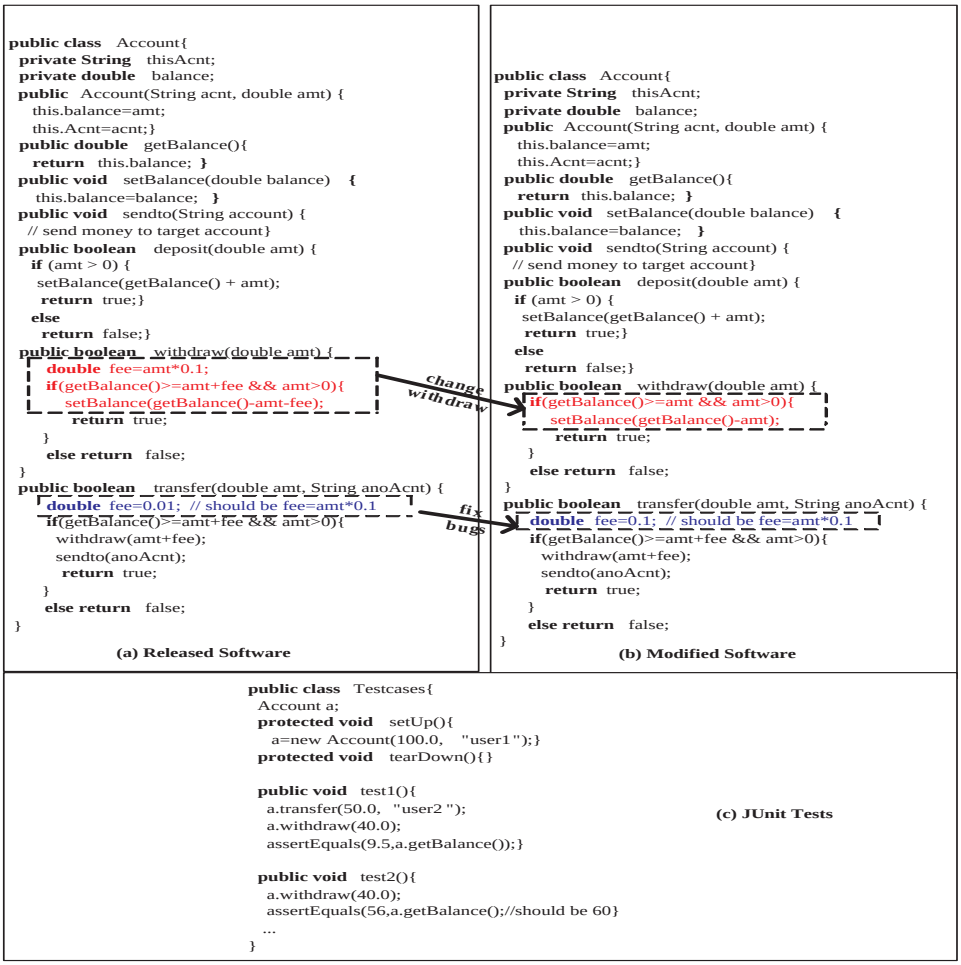

Fig. 1. An Example Program and Its JUnit Tests

regression testing process but also can help reduce the cost of code modification to fix bugs or repair tests. If the failure is due to a bug, developers can use automated techniques (e.g., [52, 56]) to decrease the cost of debugging in the product code; if the failure is due to an obsolete test, developers can also use automated techniques (e.g., [6–8]) to decrease the cost of test repair in the test code. It should be noted that techniques for automated debugging typically assume tests to be correct and focus on the product code to fix bugs. Similarly, techniques for automated test repair typically assume that the cause of the failure lies in the obsoleteness of tests. Therefore, it becomes the preceding condition for developers to identify the cause of the failure when they observe a failure [41]. That is to say, understanding the cause of a failure actually serves as an indicator for whether to apply automated debugging techniques or automated test-repair techniques.

In this paper, we propose a novel approach to classifying the causes of regression test failures for Java programs using JUnit as the framework for regression testing. In particular, we transform the problem of classifying the cause (i.e., buggy product code or obsolete test code) of a regression test failure into a problem of learning a classifier based on the data of various features related to failures. Specifically, our approach adopts the Best-first Decision Tree Learning algorithm [47, 48], which is one of the typical machine-learning algorithms in the literature. Moreover, we collect the data of the failure related features used for classification via analyzing the software under test and its test code.

To evaluate our machine-learning based approach, we performed three empirical studies on two Java programs: Jfreechart and Freecol. The first study aims to evaluate whether our approach is effective when being applied for the same version of a program. That is, the training set and the testing set consist of instances from the same version of a program. The second study aims to evaluate whether our approach is effective when being applied between versions of a program. That is, the training set and the testing set are instances of different versions of a program. The third study aims to evaluate whether our approach is effective when being applied between different programs. That is, the training set and the testing set are instances of different programs. According to the results of our empirical studies, our approach is effective in correctly classifying the causes of regression test failures when it is applied within the same program (including the same version and different versions).

In summary, this paper makes the following main contributions.

- First, we present a machine-learning based approach to classifying the causes of regression test failures. To our knowledge, this is the first piece of research that tries to classify the cause of a regression test failure as a bug in the product code or an obsolete test.
- Second, we performed three empirical studies to evaluate the effectiveness of the proposed approach in three scenarios: being applied within the same version, being applied within different versions of a program, and being applied between different programs.

The rest of this paper is organized as follows. Section 2 summarizes the related work. Section 3 illustrates the problem in this paper by an example. Section 4 presents the details of our approach. Section 5 presents the setup of our empirical studies. Section 6 presents the findings of the empirical studies. Section 7 presents the discussion and Section 8 concludes.

2 Related Work

To our knowledge, the work presented in this paper is the first approach to classifying the causes of regression test failures in software evolution. The research most related to our work is fault repair, including debugging in the product code and test repair, which will be discussed in Section 2.1 and Section 2.2. Our work is also related to regression testing as our work deals with tests in regression testing, and thus we will discuss regression testing techniques in Section 2.3. Furthermore, our work can be viewed as an application of machine-learning and

thus we will discuss application of machine-learning in software quality engineering in Section 2.4.

2.1 Debugging in Product Code

Software debugging focuses on identifying the locations of faults and then fixing the faults by replacing the faulty code with the correct code.

Most existing research [2, 22, 52, 64] on software debugging focuses on the first step, which is fault localization. Typically, spectrum-based fault-localization approaches [20, 27, 34] compare the execution information of failed tests and that of passed tests to calculate the suspiciousness of each structural units, and then localize the locations of faulty structural units by ranking the structural units based on their suspiciousness. As the effectiveness of these approaches is dependent on the test suites [1] and faults, some research focuses on improving these approaches via test selection [19] and generation [51]. Besides these spectrum-based fault localization techniques, some researchers transform fault localization to other mathematical problems, like the maximal satisfiability problem [28] and the linear programming problem [9]. To fix bugs, several techniques [17, 56] have been proposed to automate patch generation. For example, Weimer et. al [56] proposed to fix faults by using genetic programming to generate a large number of variants of the program. However, as these techniques may generate nonsensical patches, Kim et al. [29] proposed a patch generation approach (i.e., PAR), which uses fix patterns learned from existing human-written patches.

The research on fault localization is related to our work because potentially these techniques may be extended to solve our problem. To verify the effectiveness of these techniques on classifying the causes of regression test failures, we have conducted a preliminary experiment and found that direct application of these fault-localization techniques can hardly correctly classify the causes of regression test failures. Details of this experiment are presented in Section 7.

2.2 Test Repair

When changing requirements invalidate existing tests, tests are broken. Besides deleting obsolete tests [69] or creating new tests to exercise the changes [57], test-repair techniques [58] are proposed to repair broken tests rather than removing or ignoring these tests. Specifically, Galli et al. [15] proposed a technique to partially order broken unit tests rather than arbitrary order, according to the sets of methods these tests called. Daniel et al. [8] presented a technique (i.e., ReAssert) based on dynamic analysis and static analysis to find repairs for broken unit tests by retaining their fault-revealability. As ReAssert cannot repair broken tests when they have complex control flows or operations on expected values, Daniel et al. [6, 7] presented a novel test-repair technique based on symbolic execution to improve ReAssert to repair more test failures and provide better repairs. These techniques aim to repair broken unit tests in general, while some techniques have been proposed to repair broken tests in graphical user interfaces [36] or web applications [24].

To learn whether existing test-repair techniques are applicable in real practice, Pinto et al. [41] conducted an empirical study on how test suites evolved and found that test repair does occur in practice.

Existing research on debugging in product code and test repair (including those introduced by Section 2.1 and Section 2.2) assumes that developers have known whether the failure to be due to the product code or the test code. That is, when a failure occurs, developers may have to manually determine the cause of this failure before applying existing techniques on debugging in the product code or on test repair. Without such knowledge, developers risk to locate the faults in the wrong places. Unfortunately, to our knowledge, except our work reported in this paper, there is no previous study in the literature on this issue.

2.3 Regression Testing

Regression testing [33, 66] is a testing process, whose aim is to assure the quality of a program after modification. After a program is modified, developers often reuse existing tests for the program before modification and may add some tests for the modification. As it is time-consuming to run the aggregated tests, many test selection and/or reduction techniques [5, 21, 65, 71] have been proposed to reduce the number of tests used in regression testing. To optimize the cost spent on regression testing, test prioritization techniques [61, 62, 68] have been proposed to schedule the execution order of tests. Most research in test selection, reduction and prioritization investigates various coverage criteria, including statement coverage, function coverage [12], modified condition/decision coverage [44], and so on. Other research investigates various test selection, reduction, and prioritization algorithms, including greedy algorithms [25], genetic algorithms [35], integer linear programming based algorithms [5, 21, 71], and so on.

Our work is related to regression testing, especially related to test selection. However, our work aims to determine the causes of regression test failures, whereas test selection aims to select tests that are effective in exposing faults in the modified program. Specifically, test selection aims to select tests whose output becomes obsolete for the new version, whereas our work tends to identify the tests that become obsolete and should be modified to test the new version.

2.4 Application of Machine Learning in Software Quality Engineering

It is a relatively new topic to apply machine learning to software quality engineering [4, 14, 18, 23, 42]. Brun and Ernst [4] isolated fault-revealing properties of a program by applying machine learning to a faulty program and its fixed version. Then the fault-revealing properties are used to identify other potential faults. Francis et al. [14] proposed two new tree-based techniques to classify failing executions so that the failing executions resulting from the same cause are grouped together. Podgurski et al. [42] proposed to use supervised and unsupervised pattern classification and multivariate visualization to classify failing executions with the related cause. Bowring et al. [3] proposed to apply an active learning technique to classify software behavior based on execution data. Haran et al. [23] proposed to apply the Random Forest algorithm to classify passing executions from failing ones. Wang et al. [53] proposed to apply Bayesian Networks to predict the harmfulness of a code clone operation when developers' performing copy-and-paste operation. Host and Ostvold [26] proposed to identify the

problem in method naming by using data mining techniques. Zhong et al. [70] proposed an API usage mining framework MAPO using clustering and association rule mining. Furthermore, machine-learning techniques have also been widely applied to software defect prediction [37, 30, 60].

Generally speaking, the existing research on application of machine learning in software quality engineering mostly aims to automate fault detection or identify failing executions, whereas our work aims to identify whether faults are in the product code or in the test code.

3 Motivating Example

In software evolution, if developers' intention (e.g., changing functionality) incurs specification changes when making modifications, such failures may indicate obsolete test code. If developers' intention does not incur specification changes when making modifications, such failures probably indicate faulty product code. As code changes do not explicitly reflect developers' intention, it is challenging to determine through automatic program analysis whether an observed failure in regression testing is due to bugs in the product code or obsolete tests.

Fig. 1 presents an example Java class *Account* (including the version before modification and the version after modification) and its JUnit tests. The former version of *Account* is shown by Fig. 1(a), which contains a bug in method $transfer$. Fig. 1(c) gives its JUnit tests, including two tests (at the test-method level) $test1$ and $test2$.

When developers run this version of *Account* with the two tests, $test1$ fails but $test2$ passes. To fix the bug in *Account* that causes the failure of $test1$, developers may modify method $transfer$, shown by Fig. 1(b). In using this version of class *Account*, the bank wants to remove extra fees when consumers withdraw their savings, and thus developers have to modify method $withdraw$ of *Account*. In summary, when evolving *Account* from Fig. 1(a) to Fig. 1(b), developers modify two methods of *Account* due to different reasons. Method $withdraw$ is modified because developers change their expectations on the behavior of $withdraw$, whereas method $transfer$ is modified because developers want to fix a bug in the method.

After modification, two failures are observed when developers run the modified software in Fig. 1(b) with $test1$ and $test2$. The failure for $test1$ is due to modified *Account*, whereas the failure for $test2$ is due to the obsoleteness of this test. That is, the two failures have different causes (i.e., either in the product code or in the test code), resulting from developers' different intentions. As it is hard to automatically induce developers' intention based on only the software and its tests, it is not straightforward to tell whether an observed failure is due to the product code or the test code. That is, it is a challenging problem to classify the cause of an observed failure in software evolution. To our knowledge, our work is the first research that tries to solve this problem.

4 Approach

Despite the difficulty of our target problem, there are still some clues. For example, the complexity of the source code, the change information between versions,

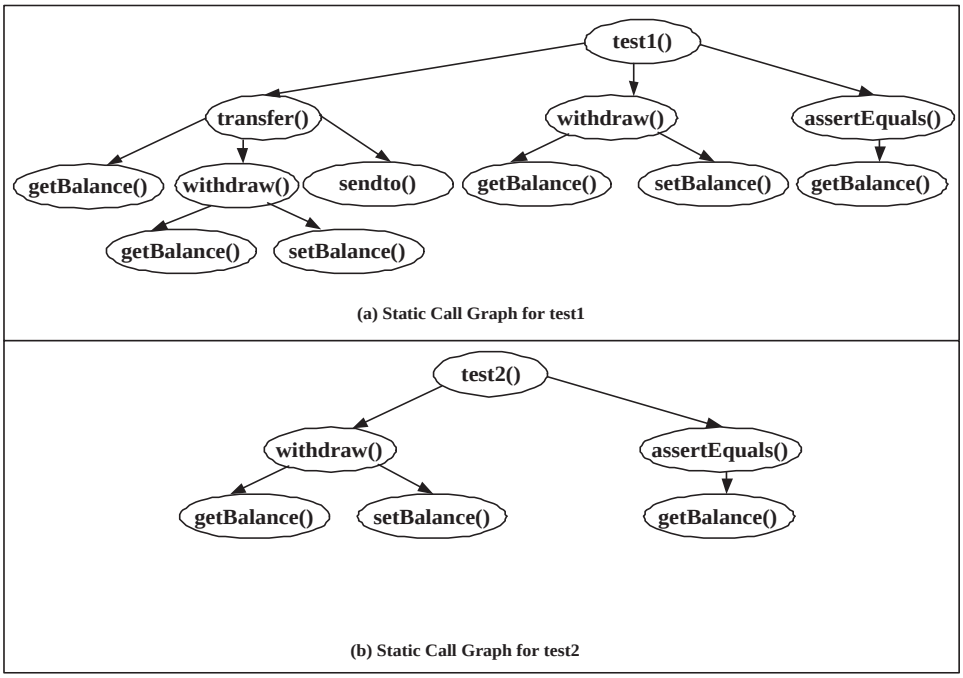

Fig. 2. Static Call Graph for the Tests

and testing information of the regression test failures may be related to the cause classification of a regression failure. Specifically, as developers are more likely to make mistakes in complex code, the failure for a regression test whose product code is complex is more likely due to bugs in the product code than the obsolete test. For example, as shown by Fig. 2, which shows the static call graphs of *test*1 and *test*2, the product code tested by *test*1 is complex as many methods of *Account* are called during the execution of *test*1, and thus the failure for *test*1 is probably due to bugs in the product code. Furthermore, more frequently changed and less tested product code is intuitively to be fault-prone. Since it is difficult to obtain some discriminative rules to classify the cause of an observed failure based on these clues, we present a machine-learning based technique to learn a classifier by using the failures whose causes are known.

In the following, we first give an overview of the proposed approach in Section 4.1. Then we present the features that we extract from the software under test and its test code in Section 4.2. Finally, we present how we train the classifier and use the classifier to classify regression test failures in Section 4.3.

4.1 Overview

In our approach, we view the problem of classifying the cause of a regression test failure as a problem of learning a classifier based on the data of failure-related features, which can be extracted and collected by analyzing the software under test and its test code. Specifically, we adopt a typical machine-learning algorithm, the Best-first Decision Tree Learning algorithm [47, 48].

In existing software development environment, when a failure occurs in the execution of a test, there is no place to record whether the cause of the failure is due to the product code or the obsolete test. To collect failures for training our classifier, our approach requires developers to record the cause of each failure when they resolve a regression test failure. After labeling the cause of each failure, our approach assigns values to seven failure-related features by statically and dynamically analyzing the software under test and its test code. Using the failures with their causes and failure-related features, our approach trains a classifier. Finally, the trained classifier can be used to classify future failures.

4.2 Identified Features

Our approach uses a set of features related to regression test failures to identify whether the cause of a regression test failure is due to the product code or the test code. Specifically, we use seven features, which can be further divided into three categories (i.e., two complexity features, one change feature, and four testing features).

Complexity Features. The complexity features are concerned with how complex the interaction between the test and the software under test is. Intuitively, the more complex the interaction is, the more likely the test can reveal a bug in the software under test. In other words, the more complex the interaction is, the more likely the failure is due to a bug in the product code.

As our target is Java programs using the JUnit testing framework, we are able to extract complexity features via statically analyzing the call graph of each test. As a JUnit test is a piece of executable source code containing testing content (i.e., a sequence of method invocations), we can determine what methods each JUnit test invokes and use such information to construct static call graphs for JUnit tests.

Given a JUnit test that induces a regression test failure, we consider the following complexity features, which are defined based on the call graph of this test at the level of methods.

- Maximum depth of the call graph (abbreviated as $MaxD$). $MaxD$ represents the length of the longest chain of nested method calls of the JUnit test. Intuitively, the larger $MaxD$ is, the more complex the interaction between the test and the software under test is, and thus the more likely the failure is due to a bug in the product code.
- Number of methods called in the graph (abbreviated as $MethodNum$), which is the total number of methods called directly or indirectly by the JUnit test by counting each method only once. Intuitively, the larger $MethodNum$ is, the more likely the failure is due to a bug in the product code.

Change Feature. The change feature is concerned with the change between the current version and its previous version of the software under test. Intuitively, if the software under test undergoes a substantial revision, it is more likely that developers want to change some behavior of the software and thus a regression test failure is more likely due to the obsoleteness of the tests; but if the software

under test undergoes a very light revision, it is more likely that developers do not want to change the behavior of the software and thus a regression test failure is more likely due to the product code. To consider this factor, we use the following change feature in our approach.

- File change (abbreviated as $FileChange$), which denotes the ratio of modification on the files that contain the methods called directly or indirectly by the failure inducing test. For a given test t, we use set $F(t)$ to represent the set of files that contain the methods called directly or indirectly by t. For a file f belonging to $F(t)$, we denote its previous version before the revision as f_b and its latter modified version as f_l. Furthermore, we use $|f_b|$ and $|f_l|$ to represent the number of lines of executable code (i.e., without counting lines containing only comments and/or blanks) of f_b and that of f_l, respectively. We then use $Change(f_b, f_l)$ to denote the number of different lines of executable code between f_b and f_l. Specifically, many tools (e.g., the unix command "Diff") can be used to compare two files and generate the different lines. Finally, we calculate $FileChange$ using the equation in Formula 1. Intuitively, the more total changes are involved in the call graph of a failure inducing test, the more likely the failure is due to an obsolete test.

$$FlieChange(t) = \frac{\sum_{\forall f \in F(t)} Change(f_b, f_l)}{\sum_{\forall f \in F(t)} maximum\{|f_b|, |f_l|\}} \quad (1)$$

Testing Features. The testing features are concerned with the testing results of all the executed tests. By using these features, our approach is able to consider the testing results of the whole test suite. In particular, we consider the following four testing features.

- Type of failure (abbreviated as $FailureType$), which denotes the type of the failing testing results returned by JUnit. In particular, $FailureType$ can be $Failure$, $Compile\_Error$ or $Runtime\_Error$, where $Failure$ denotes an assertion that is broken, $Compile\_Error$ denotes compiling problem when the compiler fails to compile the source code (including the product code and the test code) and $Runtim\_Error$ denotes a runtime exception when executing the product code with the test code.
- Count of plausible nodes in the call graph (abbreviated as $ErrorNodeNum$), which denotes the number of the methods that are called by the given failure inducing test and by at least another failure inducing test. Intuitively, if the call graph of the failure inducing test contains many such methods, the cause of the failure is more likely to lie in the product code than in the test code because one obsolete test may be unlikely to cause other tests to fail.
- Existence of highly fault-prone node in the call graph (abbreviated as $FaultProneNode$), which denotes whether the given failure inducing test calls a highly fault-prone method. Specifically, we define the highly fault-prone method as the methods that are called by more than half of the failed tests. As the highly fault-prone method is likely to contain bugs, the failure of a test calling such a method is more likely due to this method. That is, if the call graph of the failure inducing test contains such highly fault-prone

methods, the cause of the failure is more likely to lie in the product code than in the test code.

– Product innocence (abbreviated as $ProInn$), which aims to measure the ratio of innocent product code involved in the call graph of the failure inducing test. For a given failure inducing test t, we use $M(t)$ to denote the set of methods called by t. Moreover, for a method m, we use $num_p(m)$ to denote the number of passed tests that call m and $num(m)$ to denote the total number of tests that call m. Then for any failure inducing test t, we calculate $ProInn$ using the equation in Formula 2, where k is the smoothing coefficient and set to be 1 in our approach. If a failure inducing test t has larger $ProInn$, most of the product code t calls is innocent (according to Formula 2), and thus the cause of the failure is more likely to be that t itself is obsolete. That is, the larger $ProInn$ is for a failure inducing test, the more likely the failure is caused by an obsolete test in the test code.

$$ProInn(t) = \prod_{\forall m \in M(t)} \frac{num_p(m) + 1}{num(m) + k} \tag{2}$$

Note that, although the collection of data for the four testing features requires executing the test suite, there is no need to instrument the product code to record coverage information. Based on the testing results returned by the JUnit testing framework, the data collection can be done through statically analyzing the call graph.

4.3 Failure-Cause Classification

We use the following steps to learn the classifier for failure-inducing tests. First, we collect the failure-inducing tests from the software repository. Second, we extract the values of the identified seven features and label each failure-inducing test. Finally, we train a classifier using the failure-inducing tests whose feature values and failure causes are known.

Collecting Failure-Inducing Tests. When a failure is observed in regression testing, developers can record the failing tests. Such information can be stored in the software repository as a part of artifacts during software development. From the repository, our approach collects these failure-inducing tests, which are the training instances used to train a failure-cause classifier in this paper.

Determining Feature Values and Failure Causes. For the seven identified features, we determine their values by analyzing the product code under test and the tests.

For the complexity features (i.e., $MaxD$ and $MethodNum$), we calculate their values using the call graph of the failure inducing test. Specifically, we use our previous work [38] and its corresponding tool Jtop [67] to implement the static call graph used in this paper.

To determine the value of $FileChange$, we first find the methods called by the failure inducing test, and then use the method signature to identify the corresponding source code in the previous version and that in the current version. After matching the two versions, we can calculate the value of $FileChange$.

The value of $FailureType$ can be directly obtained from the JUnit testing framework. To calculate the values of $ErrorNodeNum$, $FaultProneNode$, and $ProInn$, we first obtain the testing results of all tests from the JUnit testing framework, then mark methods in the call graph of the failure inducing test as either plausible or innocent, and finally calculate the three values.

To determine the cause of each regression test failure, we require developers evolving the software to label whether a regression test failure in the training set is due to the product code or the obsolete test.

Training a Classifier Based on the training set (which contains a set of training instances with their features and labels), we are able to train a classifier for our target problem. Specifically, we adopt a typical machine-learning algorithm, the Best-first Decision Tree Learning algorithm [47, 48]. The Best-first Decision Tree Learning algorithm is based on a decision tree model, which expands the "best" nodes first rather than in depth-first order used by C4.5 algorithm [43]. The "best" node is the one whose split will lead to maximum reduction impurity among all the nodes.

After training a classifier using the Best-first Decision Tree Learning algorithm, we use the classifier to classify the cause of future failure-inducing tests.

5 Experimental Studies

5.1 Research Questions

We have conducted three empirical studies to investigate the performance of the proposed approach in three scenarios.

In scenario (1), our approach constructs a classifier based on some regression test failures for a version of a program and uses the constructed classifier to classify the causes of other regression test failures for the same version. Thus, the first research question (**RQ1**) is as follows: Is our approach effective in classifying the causes of regression test failures when being applied within one version of each program?

In scenario (2), our approach constructs a classifier based on regression test failures for one version of a program and uses the constructed classifier to classify the causes of regression test failures for the subsequent version of the program. Thus, the second research question (**RQ2**) is as follows: Is our approach effective in classifying the causes of regression test failures when being applied between two versions of each program?

In scenario (3), our approach constructs a classifier based on one program and uses the constructed classifier to classify the causes of regression test failures for another program. Thus, the third research question (**RQ3**) is as follows: Is our approach effective in classifying the causes of regression test failures when being applied across different programs?

5.2 Studied Subjects

In our experimental studies, we used two non-trivial Java programs (i.e., Jfreechart and Freecol), whose product code and test code are available from SourceForge[5].

[5] http://sourceforge.net.

Table 1. Subjects in Our Studies

Program	Product Code				Test Code			
	#Files	#LOC	#Classes	#Methods	#Files	#LOC	#Classes	#Methods
Jfreechart 1.0.0	463	68,761	465	6,028	273	26,847	273	1,751
Jfreechart 1.0.7	538	80,927	540	7,335	356	42,052	356	2,634
Jfreechart 1.0.13	585	91,101	587	8,296	383	47,930	383	3,078
Freecol 0.10.3	578	94,031	579	6,757	85	13,022	85	493
Freecol 0.10.5	602	95,404	603	7,061	87	13,226	87	497

Jfreechart[6] is a Java application used to construct graphs and tables. Freecol[7] is a software game. Each of these two programs has several available versions whose test code is written in the JUnit framework. In our experimental studies, we used only the versions whose release dates are not in the same year so that the changes between versions are nontrivial.

Table 1 depicts the statistics of these two programs. Specifically, the first four columns depict the number of files, the total number of lines of executable code (by removing comments and blanks), the number of classes, and the number of methods in the product code, whereas the latter four columns depict the number of files, the total number of lines of executable code, the number of classes, and the number of methods in the test code.

As our approach is proposed to classify the cause of a regression test failure as a bug in the product code or an obsolete test, it is necessary to collect these two types of failures from practice.

Developers usually release a version of the software when its tests cannot reveal any major faults in this version. Therefore, it is difficult to obtain faults that cause the regression test fail. Thus we manually injected faults in the product code by following some standard procedures [20]. Specifically, we randomly selected statements scattered in different files of the product code for each program and then generated faults by using mutant operators including negating a decision in conditional statements like "if" and "while", changing the values of some constants, and so on. After applying the test suite to the faulty product code, more failures would appear. We viewed the failures caused by the injected faults as due to bugs in the product code.

It is accessible to collect obsolete tests in practice. For either Jfreechart or Freecol, developers have released several versions during its development (as shown by Table 1). In regression testing, the tests for the old version may become obsolete for the new version and thus developers may need to modify existing tests or add new tests. Therefore, to access the obsolete tests in practice, we applied the test suite for a previous version of each program to the product code of the current version of the program. As the version of the used test suite is not consistent with the version of the product code, some tests may fail due to test obsoleteness. Here, we also refer to obsolete tests as faults in the test code, since they represent defects of the used test suite.

We summarize the statistical information of the faults in both product code and test code in Table 2, in which the first column depicts the abbreviation of

[6] http://www.jfree.org/jfreechart/.
[7] http://www.sourceforge.net/projects/freecol.

Table 2. Faults in Our Studies

Abbreviation	Program	Test Suite	#Tests	# Faults in Test	# Faults in Product
J1	Jfreechart 1.0.7	Jfreechart 1.0.0	1,037	9	17
J2	Jfreechart 1.0.13	Jfreechart 1.0.7	1,706	8	18
F	Freecol 0.10.5	Freecol 0.10.3	362	82	74

the program, the following two columns depict the product code and the test code, and the latter three columns depict the number of tests in each test suite, the number of faults in the test code, and the number of faults in the product code.

5.3 Experimental Design

After preparing faulty product code and faulty test code, we collected the values of features of each program by analyzing the call graphs constructed by Jtop[8] and the testing results returned by JUnit. Based on the above data, we performed the following three studies.

In our first study, we constructed a classifier and evaluated the effectiveness of the constructed classifier by using the instances from the same version of a program (i.e., Jfreechart 1.0.7, Jfreechart 1.0.13, or Freecol 0.10.5). For each version of a program, we used all its collected regression test failures, which are labeled by their causes (i.e., faults in the product code or faults in the test code), as instances and randomly split all the instances into a training set and a testing set. To reduce the influence of random selection, we used the 10 fold cross-validation technique to implement the selection process. Specifically, for a given set of data whose number of instances (i.e., regression test failures) is M, the 10 fold cross-validation technique divides the given set into 10 subsets of data equally so that each subset has $\frac{M}{10}$ instances. Then the 10 fold cross-validation technique uses each of these subsets as a testing set and the other subsets as a training set. In other words, the 10 fold cross-validation technique randomly selects $\frac{M*9}{10}$ instances of the given set as a training set, and takes the rest instances as a testing set. Moreover, the preceding process has been repeated 10 times within the 10 fold cross-validation technique. Based on each training set, our approach generates a classifier, which is evaluated by the instances of the testing set. Specifically, the Best-first Decision Tree Learning algorithm used in our approach is implemented on Weka 3.6.6[9], which is a popular environment for knowledge analysis based on machine learning and supports most machine-learning algorithms.

In our second study, we constructed a classifier by using the instances from a version of a program (i.e., Jfreechart 1.0.7) and evaluated the effectiveness of the constructed classifier by using the instances from the subsequent version of the program (i.e., Jfreechart 1.0.13). Specifically, we used all the regression test failures of Jfreechart 1.0.7 as the training set and all the regression test failures

[8] Jtop is a test management tool built in our previous research and is accessible at http://jtop.sourceforge.net/.

[9] http://www.cs.waikato.ac.nz/ml/weka/.

of Jfreechart 1.0.13 as the testing set. Based on the training set, our approach generates a classifier, which is evaluated by the instances of the testing set.

In our third study, we constructed a classifier by using the instances from one program (i.e., both the two versions of Jfreechart (including 1.0.7 and 1.0.13) or Freecol 0.10.5) and evaluated the effectiveness of the constructed classifier by using the instances from another program (i.e., Freecol 0.10.5 or both versions of Jfreechart). That is, we used all the regression test failures of one program as a training set and all the regression test failures of another program as a testing set. Based on each training set, our approach generates a classifier, which is evaluated by the instances of the testing set.

For each failed test in the testing set, we recorded (1) the number of failures correctly classified as faults in the product code, and (2) the number of failures correctly classified as faults in the test code. We then calculated the values of the metrics in Section 5.4 to evaluate our approach.

5.4 Evaluation Metrics

We used the following metrics to evaluate our approach.

- $OverAcc$, which denotes the overall accuracy of the proposed approach. This metric measures the likelihood of our approach to make a correct classification considering both causes of regression test failures.
- $AccFT$, which denotes the accuracy of classifying regression test failures due to faults in the test code. This metric measures the likelihood of our approach to make a correct classification considering only regression test failures due to faults in the test code.
- $AccFP$, which denotes the accuracy of classifying regression test failures due to faults in the product code. This metric measures the likelihood of our approach to make a correct classification considering only regression test failures due to faults in the product code.

An ideal approach should achieve values close to 1 for the $OverAcc$ metric. Furthermore, for an ideal approach, the values of $AccFT$ and the values of $AccFP$ should not differ very much. It is not complete and reliable to compare two classification approaches based on only $AccFT$ or $AccFP$. For example, supposing that there are totally 50 obsolete tests in the test code and 50 faults in the product code, if we classify all of them to be obsolete tests in the test code, the $AccFT$ is 100% but its $AccFP$ is 0% and its $OverAcc$ is 50%. This approach is obviously bad as a good approach should have a high $OverAcc$ with balanced $AccFT$ and $AccFP$.

5.5 Threats to Validity

The threat to construct validity comes from the tests whose failures are due to the product code. It is hard to collect bugs in the product code in software evolution since developers usually release a software product after fixing bugs exposed by tests. To reduce this threat, we manually inject faults in the product code following the standard procedure [20] in software testing and debugging. The standard procedure is similar to mutant generation. We did not use existing

mutation tools (e.g., MuJava[10]) to generate mutants because such tools usually generate a very large number of mutants [63]. The threat to internal validity comes from our implementation. To reduce this threat, we reviewed all the code before conducting our experiments. The threats to external validity lie in the subjects used in the studies and the impact of machine learning. In this paper, we used five versions of two open source Java programs, which are not necessarily representative of other programs. As we considered three scenarios, the overall workload of evaluation for our research can be already similar to or even more than that for an existing piece of research on test repair, from which our research stems. To further reduce the threat from subjects, we need to evaluate our approach on larger programs in other language (e.g., C#, C++) with more failure-inducing tests. Another external threat comes from machine learning, as our approach constructs a classifier by applying some existing machine-learning algorithm to some subjects. Although machine learning does not have the power to identify the cause-effect chain to pinpoint the differentiator between the product code and the test code, it is a highly generalizable way to generate a solution to pinpoint the preceding differentiator. For example, if we construct a solution based on some observations, the solution can be applicable to only subjects for which the observations hold. However, as a machine-learning-based approach has the ability to summarize observations from existing data, we reply on machine learning instead of inventing a classifier manually. Note that there may not necessarily be just one same classifier for all different subjects.

6 Results and Analysis

In this section, we first present the results and analysis of the three studies in Section 6.1, Section 6.2, Section 6.3, then give a sample of decision tree in Section 6.4, and finally summarize the main findings of our experimental studies in Section 6.5.

6.1 Study I – Within the Same Version

Fig. 3 presents the results of our first study. Specifically, the top sub-figure depicts the overall accuracy of classification results (i.e., $OverAcc$), whereas the bottom sub-figures depict the accuracy of classification failures due to faults in the test code (i.e., $AccFT$) and the accuracy of classification failures due to faults in the product code (i.e., $AccFP$). For simplicity, we use J1, J2, and F to represent Jfreechart 1.0.7, Jfreechart 1.0.13, and Freecol 0.10.5. In study I, the training instances and testing instances are collected from the same version of a program (i.e., version 1.0.7 of Jfreechart using the test suite of version 1.0.0, version 1.0.13 of Jfreechart using the test suite of version 1.0.7, version 0.10.5 of Freecol using the test suite of version 0.10.3).

Concerning the comparison of the overall accuracy ($OverAcc$) of our approach with 50%, which can be regarded as a random classification or a blind guess of failure causes, all of our approach's $OverAcc$ values for the programs are are around 80%, much larger than 50%. Furthermore, the values of $AccFT$ and $AccFP$ are usually close to the corresponding values of $OverAcc$ except for

[10] http://cs.gmu.edu/∼offutt/mujava/.

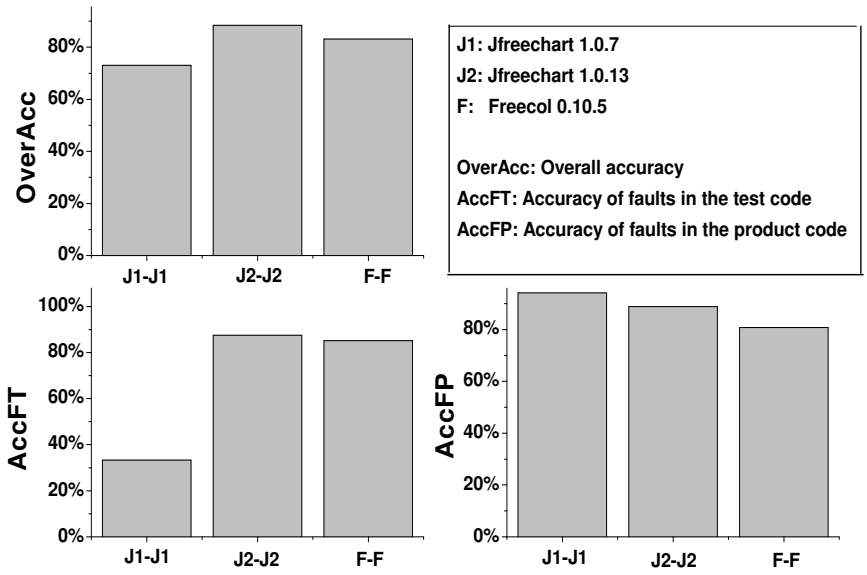

Fig. 3. Results of Study I

Jfreechart 1.0.7. That is to say, the classifier of our approach constructed by using one version of a program can usually classify the causes of regression test failures of the same version quite accurately. Our approach is not very effective in classifying the faults in the test code of Jfreechart 1.0.7. We suspect the reason to be that Jfreechart 1.0.7 has a small number of tests (test suite of version 1.0.0) and a small number of faults in the test code so as to bias the classification results.

6.2 Study II – Between Versions

Fig. 4 depicts the results of our second study. The results for Jfreechart 1.0.7 are based on the training instances collected from version 1.0.7 of Jfreechart (using the test suite of version 1.0.0) and the testing instances collected from version 1.0.13 of Jfreechart (using the test suite of version 1.0.7).

The *OverAcc* value of our approach is higher than 50%, which is 96.15%. That is to say, the classifier constructed using some version of a program may be used to classify the cause of a regression test failure of another version of the program. Moreover, the results *AccFT* and *AccFP* for Jfreechart 1.0.7 are both close to 100%.

Furthermore, comparing the classification results of Jfreechart 1.0.13 (whose training instances and testing instances are all from Jfreechart 1.0.13) in Fig. 3 and those (whose training instances are from Jfreechart 1.0.7 but testing instances are from Jfreechart 1.0.13) in Fig. 4, the results (including *OverAcc*, *AccFT*, and *AccFP*) of the program increase. Although the differences between versions may harm the accuracy of a classifier, more instances are used to train the classifier in study II than study I, and thus our approach produces better results in study II than in study I.

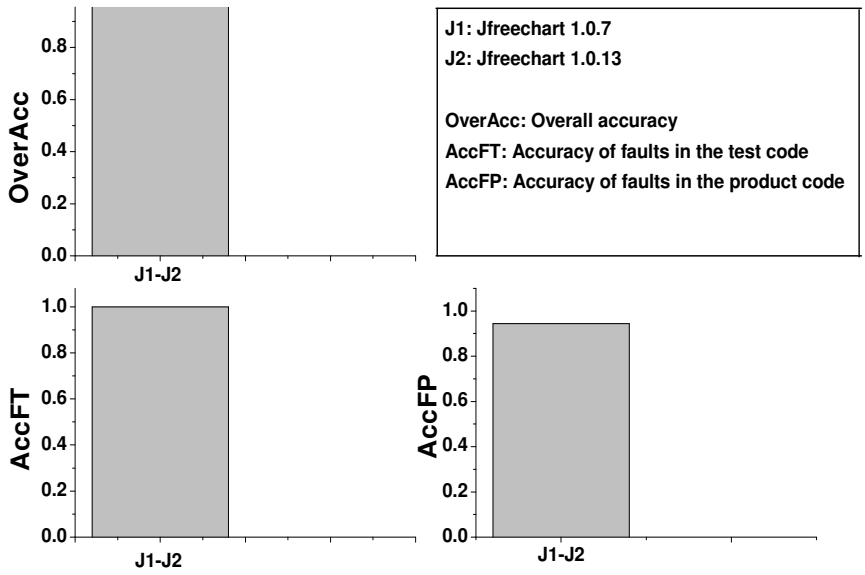

Fig. 4. Results of Study II

6.3 Study III – Across Programs

Fig. 5 depicts the results of our third study, where J is the abbreviation of Jfreechart (including its 1.0.7, and 1.0.13). The training instances are from version 1.0.7 and version 1.0.13 of Jfreechart and the testing instances are from version 0.10.5 of Freecol, or vice versa.

The *OverAcc* values of our approach are still higher than 50%, which are 68.18% and 71.15%, respectively. That is to say, our approach can still provide some help over the baseline. However, comparing to the classification results in Fig. 3 and Fig. 4, there are significant decreases. Furthermore, when closely examining the *AccFT* results and the *AccFP* results, the former may not be acceptable, because the values of *AccFT* are too low. In other words, the approach classifies too many failures as faults in the product code. We suspect the reason to be that the two programs used in the training set and in the testing set have significant differences in their structures so that the training process may have to face many noises.

The results of our third study are not satisfactory enough to be applied in practice, but it indicates the possibility that our approach may be applied between programs by improving the classifier using more features. It should be noted that cross-program validation is notoriously difficult for mining based approaches [53, 72]. One possible way to alleviate this drawback is to set up a multi-program training set to prevent the trained classifier from being too specific to one program.

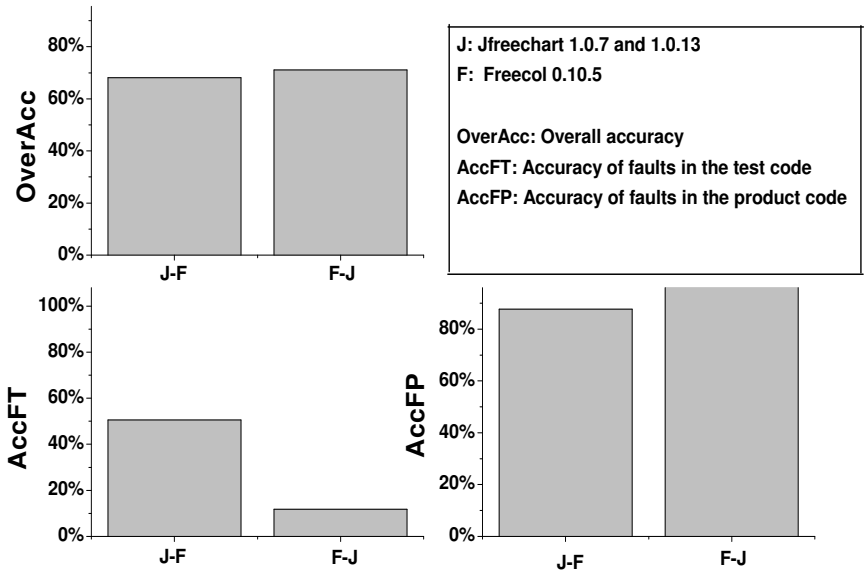

Fig. 5. Results of Study III

6.4 A Decision Tree Sample

Fig. 6 presents a decision tree that are constructed using the all the regression test failures of Jfreechart 1.0.7 as training instances. In this decision tree, only three features (i.e., $FileChange$, $MaxD$, and $ErrorNodNum$) have contribution to differentiate the faults in the test code and the faults in the product code. In particular, if the values of $FileChange$ are smaller than 9.0, the causes of regression test failures are classified as faults in the product code; if the values of $FileChange$ are no smaller than 9.0 and the values of $MaxD$ are no smaller than 11.5, the causes of regression test failures are also classified as faults in the product code; if the values of $FileChange$ are no smaller than 9.0 and the values of $MaxD$ are smaller than 11.5 and the values of $ErrorNodeNum$ are smaller than 8.5, the causes of regression test failures are classified as faults in the test code. If the values of $FileChange$ are no smaller than 9.0 and the values of $MaxD$ are smaller than 11.5 and the values of $ErrorNodeNum$ are no smaller than 8.5, the causes of regression test failures are not clear and may be classified as faults in the product code or in the test code. We did not present the decision trees generated in the three studies due to the large number of generated decision trees.

6.5 Summary

In summary, the main findings of our experimental studies are as follows.

– First, our approach produces acceptable results when the training instances and the testing instances are from the same version of one program or from the different versions of one program.

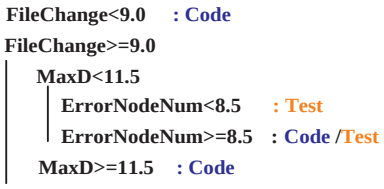

Fig. 6. Visualization of A Decision Tree

– Second, when the training instances and the testing instances are from different programs, our approach is not as effective as being applied in the same program.

According to the two findings, our machine-learning based approach is generally effective to classify causes of regression test failures when the training instances and the testing instances are from the same program (including different versions and the same versions).

7 Discussion

In this section, we will discuss some issues that are related to our approach.

Adjustment of Features. Although our approach uses only seven features to construct a classifier, more features may be added to improve the effectiveness of the approach. Specifically, the complexity features and the change feature are collected by static analysis, whereas the testing features are collected by lightweight dynamic analysis. Dynamically collected features and statically collected features are complementary, and we will consider their cost and effectiveness in constructing a classifier in our future work. Furthermore, the seven features are intuitively correlated with the cause of a regression test failure, but our experimental studies did not investigate the impact of each feature on the classification results. Moreover, our intuition that how each feature impacts the classification results (e.g., a regression test failure is more likely due to buggy product code if the corresponding test has a small value of $FileChange$) is not encoded in the classifier since our intuition may not be consistent with the actual practice. To learn explicitly the contribution of each feature on classification results, we will conduct more experiments on various combination of these features in the future.

Applicability of Fault Localization Techniques. Our preliminary experimental results show that existing fault localization approaches can hardly be directly used to solve the problem of this paper. Conceptually, fault localization, especially spectrum-based fault localization approaches may be extended to solve the problem in this paper. Therefore, we conducted a preliminary experimental study on two programs (with the same seeded faults) and their test suites in Table 2 using Tarantula [27], which is an effective and widely used spectrum-based fault localization approach. For each program and its test suite, Tarantula generates a ranked list of suspicious methods based on the descendent order of

Table 3. Results of Tarantula in Classifying Test Failures

Program −Test Suite	Faults in the Test Code				Faults in the Product Code			
	#Total	#Correct	#Wrong	#Miss	#Total	#Correct	#Wrong	#Miss
Jfreechart 1.0.7 − Jfreechart 1.0.0	9	4	2	3	17	0	2	15
Jfreechart 1.0.13 − Jfreechart 1.0.7	8	1	7	0	16	15	0	1
Freecol 0.10.5 − Freecol 0.10.3	82	0	14	68	74	0	1	73

their suspiciousness, which measures the possibility that the methods contain faults. Formally, the ranked list of suspicious methods is denoted as m_1, m_2, ..., m_n, where m_i and m_j $(1 \leq i, j \leq n)$ denotes any methods of the program and its test suite. Supposed that $sus(m_i)$ and $sus(m_j)$ represent the suspiciousness of methods m_i and m_j, respectively, then $sus(m_i) \leq sus(m_j)$ iff $j \leq i$. If m_1 is a method in the product code and any method m_k in the test code satisfying that $sus(m_1) > sus(m_k)$, we deem that the faults are in the product code; if m_1 is a method in the test code and any method m_k in the product code satisfying that $sus(m_1) > sus(m_k)$, we deem that the faults are in the test code; otherwise, it is not clear where the faults are since both the method in the product code and the method in the test code have the largest suspiciousness. Based on this assumption, the classification results of Tarantula can be summarized as Table 3, where #Total shows the total number of faults in either the test code or the product code, #Correct shows how many faults (in either the test code or the product code) have been correctly classified by Tarantula, #Wrong shows how many faults have been incorrectly classified by Tarantula (i.e., the faults in the test code have been classified as faults in the product code, or the faults in the product code have been classified as faults in the test code), and #Miss shows how many faults (in either the test code or the product code) cannot be clearly[11] classified by Tarantula. According to this table, Tarantula can hardly precisely classify the cause of a failed test in most cases, especially when the faults are due to an obsolete test. Therefore, existing fault localization approaches cannot be directly applied to solve the problem in this paper. In the future, we will consider how to utilize the results of fault localization approaches to improve the classification.

Impact of Structure Changes. To collect the change feature, our approach is required to identify changes between versions, but some structural changes (e.g., renaming) may cause noise in matching the entities between versions. Changes on the program's behavior should be manifested on its test case, whereas changes only on the program's structure may not. Therefore, it is important to precisely map the entities between versions so as to reduce the noise in gathering the change feature. Refactoring [11, 49] is an important type of changes in object-oriented software, which changes the structure of a program without affecting its behavior [13]. As there exist many refactoring tools [10, 55], we will use these

[11] Sometimes methods in the product code and methods in the test code are assigned with the same suspiciousness and thus Tarantula cannot tell whether the faults are due to the test code or the product code.

tools to match entities between versions in collecting the change feature so as to reduce the noise resulting from some structure changes in the future.

Other Machine-Learning Algorithms. In this paper we present and evaluate our approach using a typical machine-learning algorithm (i.e., Best-first Decision Tree algorithm), but there exist many machine-learning algorithms in the literature. Besides Best-first Decision Tree algorithm, we have implemented our approach using another algorithm the Naive Bayes algorithm [39], and found that the latter was much worse than the former. Besides machine-learning algorithms, there are many other factors (e.g., size of samples) that may influence the effectiveness of the proposed approach. As this paper is only a first step on classifying the cause of a regress test failure, we will further investigate the impact of these factors in our future work.

8 Conclusions and Future Work

In software evolution, when we apply an existing test suite to the modified software, some regression tests may fail. The failures of these regression tests may be due to buggy product code or obsolete test code. Before applying existing debugging techniques in the product code or test-repair techniques, it is necessary to determine whether a failure is due to the bug in the product code or obsolete tests. In this paper, we propose a machine-learning based approach, which collects values of seven features that may be related to failures of regression tests and then constructs a classifier by using a machine-learning algorithm (i.e., the Best-first Tree Learning algorithm). Furthermore, we evaluated this approach in three scenarios and found that the overall accuracy of our approach on correctly classifying regression failures is mostly about 80% when being applied within a program.

In future, we will identify more failure-related features to further improve the classification accuracy. We will also evaluate the proposed approach on a variety of projects written in different programming languages. As the empirical study did not evaluate the effectiveness of the given features on classification, we will evaluate which features play a leading role in the classification in our future work. Furthermore, we will work on the feasibility of establishing a discriminative model, aiming at classifying the causes of regression test failures based on features.

9 Acknowledgement

This research is sponsored by the National 973 Program of China No. 2009CB320703, the National 863 Program of China No. 2012AA011202, the Science Fund for Creative Research Groups of China No. 61121063, and the National Natural Science Foundation of China under Grant Nos. 91118004, 61225007, 61228203, and 61272157.

References

1. Abreu, R., Zoeteweij, P., van Gemund, A.J.C.: On the accuracy of spectrum-based fault localization. In: Testing: Academic and Industrial Conference Practice and Research Techniques, pp.89-98 (2007)

2. Artzi, S., Kiezun, A., Dolby, J., Tip, F., Dig, D., Paradkar, A., Ernst, M.D.: Finding bugs in web applications using dynamic test generation and explicit state model checking. IEEE Transactions on Software Engineering 34(4), 474-494 (2010)
3. Bowring, J.F., Rehg, J.M., Harrold, M.J.: Active learning for automatic classification of software behavior. In: ISSTA, pp.195-205 (2004)
4. Brun, Y., Ernst, M.D.: Finding latent code errors via machine learning over program executions. In: ICSE, pp.480-490 (2004)
5. Black, J., Melachrinoudis, E., Kaeli, D.: Bi-criteria models for all-uses test suite reduction. In: ICSE, pp.106-115 (2004)
6. Daniel, B., Dig, D., Gvero, T., Jagannath, V., Jiaa, J., Mitchell, D., Nogiec, J., Tan, S.H., Marinov, D.: ReAssert: a tool for repairing broken unit tests. In: ICSE, pp.1010-1012 (2011)
7. Daniel, B., Gvero, T., Marinov, D.: On test repair using symbolic execution. In: ISSTA, pp.207-218 (2010)
8. Daniel, B., Jagannath, V., Dig, D., Marinov, D.: Reassert: suggesting repairs for broken unit tests. In: ASE, pp.433-444 (2009)
9. Dean, B.C., Pressly, W.B., Malloy, B.A., Whitley, A.A.: A linear programming approach for automated localization of multiple faults. In: ASE, pp.640-644 (2009)
10. Dig, D., Comertoglu, C., Marinov, D., Johnson, R.: Automated detection of refactorings in evolving components. In: ECOOP, pp. 404-428 (2006)
11. Dig, D., Marrero, J., Ernst, M.D.: Refactoring sequential Java code for concurrency via concurrent libraries. In: ICSE, pp. 397-407 (2009)
12. Elbaum, S., Malishevsky, A.G., Rothermel, G.: Test case prioritization: a family of empirical studies. IEEE Transactions on Software Engineering. 28(2), 159-182 (2002)
13. Fowler, M., Beck, K., Brant, J., Opdyke, W., Roberts, D.: Refactoring: Improving the Design of Existing Code. Addison-Wesley (1999)
14. Francis, P., Leon, D., Minch, M., Podgurski, A.: Tree-based methods for classifying software failures. In: ISSRE, pp.451-462 (2004)
15. Galli, M., Lanza, M., Nierstrasz, O., Wuyts, R.: Ordering broken unit tests for focused debugging. In: ICSM, pp.114-123 (2004)
16. Ghezzi, C., Jazayeri, M., Mandrioli, D.: Fundamentals of Software Engineering. Prentice Hall PTR (2002)
17. Goues, C.L., Nguyen, T., Forrest, S., Weimer, W.: GenProg: A generic method for automatic software repair. IEEE Transactions on Software Engineering 38(1), 54-72 (2012)
18. Hao, D., Wu, X., Zhang, L.: An empirical study of execution-data classification based on machine learning. In: SEKE, pp. 283-288 (2012)
19. Hao, D., Xie, T., Zhang, L., Wang, X., Mei, H., Sun, J.: Test input reduction for result inspection to facilitate fault localization. Automated Software Engineering 17(1), 5-31 (2010)
20. Hao, D., Zhang, L., Pan, Y., Mei, H., Sun, J.: On similarity-awareness in testing-based fault-localization. Automated Software Engineering 15(2), 207-249 (2008)
21. Hao, D., Zhang, L., Wu, X., Mei, H., Rothermel, G.: On-demand test suite reduction. In: ICSE, pp.738-748 (2012)
22. Hao, D., Zhang, L., Xie, T., Mei, H., Sun, J.: Interactive fault localization using test information. Journal of Computer Science and Technology 24(5), 962-974 (2009)
23. Haran, M., Karr, A., Orso, A., Porter, A., Sanil, A.: Applying classification techniques to remotely-collected program execution data. In: FSE, pp.146-155 (2005)
24. Harman, M., Alshahwan, N.: Automated session data repair for web application regression testing. In: ICST, pp. 298-307 (2008)

25. Harrold, M.J., Gupta, R., Soffa, M.L.: A methodology for controlling the size of a test suite. ACM Transactions on Software Engineering and Methodology 2(3), 270-285 (1993)
26. Host, E.W., Ostvold B.M.: Debugging method name. In: ECOOP, pp. 294-317 (2009)
27. Jones, J.A., Harrold, M.J.: Empirical evaluation of tarantula automatic fault-localization technique. In: ASE, pp.273-282 (2005)
28. Jose, M., Majumdar, R.: Cause clue clauses: Error localization using maximum satisfiability. In: PLDI, pp.437-446 (2011)
29. Kim, D., Nam, J., Song, J., Kim, S.: Automatic patch generation learned from human-written patches. In: ICSE (2013) to appear
30. Kim, S., Zhang, H., Wu, R., Gong, L.: Dealing with noise in defect prediction. In: ICSE, pp.481-490 (2011)
31. Koskinen, J.: Software Maintenance Cost, http://users.jyu.fi/ koskinen/smcosts.htm (2003)
32. Lehman, M.M., Belady, L.A.: Program Evolution C Processes of Software Change. Academic Press, London (1985)
33. Leung, H.K.N., White, L.: Insights into regression testing. In: ICSM, pp.60-69 (1989)
34. Liblit, B., Aiken, A., Zheng, A.X., Jordan, M.I.: Bug isolation via remote program sampling. In: PLDI, pp.141-154 (2003)
35. Mansour, N., El-Fakih, K.: Simulated annealing and genetic algorithms for optimal regression testing. Journal of Software Maintenance 11(1), 19-34 (1999)
36. Memon, A.M.: Automatically repairing event sequence-based gui test suites for regression testing. ACM Transactions on Software Engineering and Methodology 18(2), 1-35 (2008)
37. Menzies, T., Greenwald, J., Frank, A.: Data mining static code attributes to learn defect predictors. IEEE Transactions on Software Engineering 32(11), 1-12 (2007)
38. Mei, H., Hao, D., Zhang, L., Zhang, L., Zhou, J., Rothermel, G.: A static approach to prioritizing Junit test cases. IEEE Transactions on Software Engineering 38(6), 1258-1275 (2012)
39. McCallum, A., Nigam, K.: A comparison of event models for naive bayes text classification. In: AAAI, pp. 41-48 (1998)
40. Mirzaaghaei, M., Pastore, F., Pezze, M.: Supporting test suite evolution through test case adaption. In: ICST, pp.231-240 (2012)
41. Pinto, L.S., Sinha, S., Orso, A.: Understanding myths and realities of test-suite evolution. In: FSE, pp.1-11 (2012)
42. Podgurski, A., Leon, D., Francis, P., Masri, W., Minch, M., Sun, J., Wang, B.: Automated support for classifying software failure reports. In: ICSE, pp.465-474 (2003)
43. Quinlan, J.R.: Introduction of Decision Trees. Machine Learning (1986)
44. Rajan, A., Whalen, M.W., Heimdahl, M.P.E.: The effect of program and model structure on MC/DC test adequacy coverage. In: ICSE, pp.161-170 (2008)
45. Robinson, B., Ernst, M.D., Perkins, J.H., Augustine, V., Li, N.: Scaling up automated test generation: Automatically generating maintainable regression unit tests for programs. In: ASE, pp.23-32 (2011)
46. Seacord, R.C., Plakosh, D., Lewis, G.A.: Modernizing Legacy Systems: Software Technologies, Engineering Process and Business Practices. Addison-Wesley (2003)
47. Shi, H.: Best-First Decision Tree Learning. Master Thesis, the University of Waikato (2007)
48. Stiglic, G., Kocbek, S., Kokol, P.: Comprehensibility of classifiers for future microarray analysis datasets. In: PRIB, pp.1-11 (2009)

49. Taneja, K., Dig, D., Xie, T.: Automated detection of api refactorings in libraries. In: ASE, pp. 377-380 (2007)
50. Tillmann, N., Schulte, W.: Unit Tests Reloaded: Parameterized Unit Testing with Symbolic Execution. Microsoft Research, Technical Report (2005).
51. Wang, T., Roychoudhury, A.: Automated path generation for software fault localization. In: ASE, pp.347-351 (2005)
52. Wang, X., Cheung, S.C., Chan, W.K., Zhang, Z.: Taming coincidental correctness: Coverage refinement with context patterns to improve fault localization. In: ICSE, pp.45-55 (2009)
53. Wang, X., Dang, Y., Zhang, L., Zhang, D., Lan E., Mei, H.: Can I clone this piece of code here?. In: ASE, pp. 170-179 (2012)
54. Wang, X., Zhang, L., Xie, T., Anvik, J., Sun. J.: An approach to detecting duplicate bug reports using natural language and execution information. In: ICSE, pp. 461-470 (2008)
55. Weiβgerer, P., Diehl, S.: Identifying refactorings from source-code changes. In: ASE, pp. 231-240 (2006)
56. Weimer, W., Nguyen, T., Goues, C.L., Forrest, S.: Automatically finding patches using genetic programming. In: ICSE, pp.364-374 (2009)
57. Wloka, J., Ryder, B.G., Tip, F.: JUnitMx - A change-aware unit testing tool. In: ICSE, pp.567-570 (2009)
58. Yang, G., Khurshid, S., Kim, M.: Specification-based test repair using a lightweight formal method. In: FM, pp.455-470 (2012)
59. Zaidman, A., Rompaey, B.V., Demeyer, S., Deursen, A.: Mining software repositories to study co-evolution of production & test code. In: ICST, pp.433-444 (2008)
60. Zhang, H., Zhang, X., Gu, M.: Predicting defective software components from code complexity measures. In: PRDC, pp.93-96 (2007)
61. Zhang, L., Hao, D., Zhang, L., Rothermel, G., Mei, H.: Bridging the gap between the total and additional test-case prioritization strategies. In: ICSE (2013) to appear
62. Zhang, L., Hou, S., Guo, C., Xie, T., Mei, H.: Time-aware test-case prioritization using integer linear programming. In: ISSTA, pp.213-223 (2009)
63. Zhang, L., Hou, S., Hu, J., Xie, T., Mei, H.: Is operator-based mutant selection superior to random mutant selection? In: ICSE, pp. 435-444 (2010)
64. Zhang, L., Kim, M., Khurshid, S.: Localizing failure-inducing program edits based on spectrum information. In: ICSM, pp. 23-32 (2011)
65. Zhang, L., Marinov, D., Zhang, L., Khurshid, S.: An empirical study of JUnit test-suite reduction. In: ISSRE, pp. 170-179 (2011)
66. Zhang, L., Marinov, D., Zhang, L., Khurshid, S.: Regression mutation testing. In: ISSTA, pp. 331-341 (2012)
67. Zhang, L., Zhou, J., Hao, D., Zhang, L., Mei, H.: Jtop: Managing JUnit test cases in absence of coverage information. In: ASE (Research Demo Track), pp. 673-675 (2009)
68. Zhang, L., Zhou, J., Hao, D., Zhang, L., Mei, H.: Prioritizing JUnit test cases in absence of coverage information. In: ICSM, pp. 19-28 (2009)
69. Zheng, X., Chen, M.H.: Maintaining multi-tier web applications. In: ICSM, pp.355-364 (2007)
70. Zhong, H., Xie, T., Zhang, L., Pei, J., Mei, H.: MAPO: Mining and recommending api usage patterns. In: ECOOP, pp. 318-343 (2009)
71. Zhong, H., Zhang, L., Mei, H.: An experimental study of four typical test suite reduction techniques. Information and Software Technolgoy 50, 534-546 (2008)
72. Zimmermann, T., Nagappan, N., Gall, H.: Cross-project defect prediction: a large scale experiment on data vs. domain vs. process. In: FSE, pp. 91-100 (2009)