Drift: Fine-Grained Prediction of the Co-Evolution of Production and Test Code via Machine Learning

Lei Liu
Southern University of Science and Technology
Shenzhen, China
12032476@mail.sustech.edu.cn

Sinan Wang
Southern University of Science and Technology
Shenzhen, China
wangsn@mail.sustech.edu.cn

Yepang Liu*
Southern University of Science and Technology
Shenzhen, China
liuyp1@sustech.edu.cn

Jinliang Deng
University of Technology Sydney
Sydney, Australia
jinliang.deng@student.uts.edu.au

Sicen Liu
Southern University of Science and Technology
Shenzhen, China
11910338@mail.sustech.edu.cn

Abstract

As production code evolves, test code can quickly become outdated. When test code is outdated, it may fail to capture errors in the programs under test and can lead to serious software bugs that result in significant losses for both developers and users. To ensure high software quality, it is crucial to promptly update the test code after making changes to the production code. This practice ensures that the test code and production code evolve together, reducing the likelihood of errors and ensuring the software remains reliable. However, maintaining test code can be challenging and time-consuming. To automate the identification of outdated test code, recent research has proposed Sitar, a machine learning-based method. Despite Sitar's usefulness, it has major limitations, including its coarse prediction granularity (at class level), reliance on naming conventions to discover test code, and dependence on manually summarized features to construct machine learning models.

In this paper, we address the limitations of Sitar and propose a new machine learning-based approach Drift. Drift predicts outdated test cases at the method level. It leverages method-calling relationships to accurately infer the links between production and test code, and automatically learns features via code analysis. We evaluate Drift using 40 open-source Java projects in both within-project and cross-project scenarios, and find that Drift can achieve satisfactory prediction performances in both scenarios. We also compare Drift with existing methods for outdated test code prediction and find that Drift can significantly outperform them. For example, compared with Sitar, the accuracy of Drift is increased by about 8.5%, the F1-score is increased by about 8.3%, and more importantly, the number of test cases that developers need to check is reduced by about 75%. Therefore, our method, Drift, can predict outdated test cases more accurately at a fine-grained level, and thus better facilitate the co-evolution of production and test code.

*Yepang Liu is the corresponding author. He is affiliated with both the Department of Computer Science and Engineering and the Research Institute of Trustworthy Autonomous Systems at Southern University of Science and Technology.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

Internetware 2023, August 4–6, 2023, Hangzhou, China
© 2023 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-0894-7/23/08…$15.00
https://doi.org/10.1145/3609437.3609449

Keywords

Software Evolution, Outdated Test Code, Machine Learning

ACM Reference Format:
Lei Liu, Sinan Wang, Yepang Liu, Jinliang Deng, and Sicen Liu. 2023. Drift: Fine-Grained Prediction of the Co-Evolution of Production and Test Code via Machine Learning. In *14th Asia-Pacific Symposium on Internetware (Internetware 2023), August 4–6, 2023, Hangzhou, China.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3609437.3609449

1 Introduction

The purpose of software testing is to examine software artifacts and behaviors to identify flaws, such as programming errors and usability issues. Effective testing can help developers avoid serious logic problems during software development, and enable product managers to expand or revise requirements, ultimately improving the user experience. In practice, building testing infrastructure and writing test code, such as unit test cases, are essential activities in the software development process. While software testing infrastructure is typically stable, test code is rarely static. With the continual updates of production code, existing test code may become outdated or inadequate, failing to cover new functionalities or leading to spurious failures due to inappropriate assertions. Therefore, test code should be updated simultaneously with production code, a process known as the co-evolution of production code and test code, or *production-test co-evolution* for short [21].

While production-test co-evolution is critical, keeping test code up-to-date can be challenging in practice. The main reason is that test maintenance tasks are typically performed manually, and it is non-trivial to identify specific locations of test code that require updates as production code evolves. Moreover, updating test code by adding new test cases or removing or revising inapplicable test cases is a time-consuming process. Due to these reasons, existing research [22] has found that in real-world projects, test code evolution often lags behind the evolution of production code, resulting in the prevalence of obsolete and inapplicable test code. To address this issue, researchers have been exploring automated techniques for supporting test code updates [4, 5, 8, 11, 12, 15].

Updating test code in response to production code changes essentially involves two steps: (1) determining whether the test code should be updated along with the production code and (2) identifying which new test cases should be added and which existing test cases should be revised or removed if the answer to the first step is yes. It is challenging to automate this task for two major reasons. First, the traceability links between production code and test code are rarely explicitly maintained in real-world projects [18]. Without precise knowledge of the correspondence between test code and production code, it is hard to do further analysis and reasoning. Second, whether to update a piece of test code can be determined by various factors and requires an understanding of the purpose of production code changes. Adopting pre-defined heuristic rules based on simple factors (e.g., when a production method's body changes, the corresponding test code should be updated) is generally ineffective for predicting necessary test changes.

The most recent work, SITAR [19], addresses the aforementioned challenges in the following manner. To associate test code with production code, SITAR leverages the file naming convention of Java projects [1] to map Java classes with their corresponding test classes. For example, the class `TestFoo` is likely a test class for the production class `Foo`. For predicting necessary test updates, SITAR employs machine learning algorithms to train a model based on the historical commits of a project. The trained model takes into account multiple factors, such as the changed program constructs in the production code and the complexity of the code changes, to infer whether there is a need to update a test class when the class under test is changed.

Despite its potential benefits, SITAR has several limitations. First, while it is possible to infer the relationship between test code and production code via analyzing file or method names, developers may not strictly follow conventions or suggested practices to name a file or a method. As a result, the recovered links between production code and test code are often incomplete, with some links being missed. Second, SITAR relies on a small set of manually determined features (e.g., whether the parameter list of a method is changed or whether there are changes in the conditions of `if` statements) to train its machine learning model. Such features, selected based on experience, may not be comprehensive. Third, SITAR can only perform coarse-grained prediction at the class level. Given a set of changes in a production class, SITAR only predicts whether the corresponding test class needs to be updated without providing more fine-grained results. As a test class may contain a large number of test cases, users of SITAR must still expend much effort to locate the specific test cases that need to be updated, which limits SITAR's practical adoption. In Section 2, we will provide a motivating example to further illustrate these limitations.

To address the limitations of existing works, we first conducted a large-scale empirical study based on 731 popular, well-maintained, and diversified open-source Java projects. Through statistical data analysis, we found that refining the prediction granularity of outdated test code to the method level can generally reduce the effort required for test case updates. Additionally, we observed that not all changes to production methods will necessitate test case updates, and whether updates are required may be related to the internal characteristics of the changed production methods. Regarding the association of production methods and test cases, we found that the existing naming convention method, which is used to establish traceability links between production files (classes) and test files (classes), is not suitable for finding the relationship between test cases and production methods at the method level. Instead, a more effective approach is to identify the multi-layer call relationship of the test case and then determine the traceability link relationship between the test case and the corresponding production method. In addition to the above findings, we also identified 48 method-level code features that may be related to test case updates, providing an accurate representation of the changed production code.

Based on the findings of our empirical research, we designed a fine-grained method to automatically predict outdated test cases, which we named DRIFT (fine-graineD pRedIction oF co-evoluTion). DRIFT effectively addresses the challenges of automatic prediction of outdated test cases and improves upon the limitations of existing techniques. The method is divided into three parts: data preprocessing, method-granularity feature extraction and classifier training, and accurate judgment and identification of the test cases that need to be updated. We also conducted experiments to evaluate DRIFT. Firstly, we evaluated DRIFT's prediction performance in both within-project and cross-project scenarios. The experiments revealed that DRIFT can achieve good prediction results in both within-project and cross-project scenarios, indicating that it can learn co-evolution laws from different projects and apply them to new projects to predict outdated test cases. Next, we compared the performance of DRIFT with two existing prediction methods: a rule-based prediction method [10] and SITAR [19]. Through experiments on 40 open-source Java projects, we found that DRIFT outperforms the two methods significantly. Compared to SITAR, the average prediction accuracy of DRIFT is increased by about 8.5%, the F1-score is increased by about 8.3%, and the number of test cases that developers need to check is reduced by about 75%. These results show that DRIFT can predict outdated test cases in a more accurate and fine-grained manner, reduce the workload of code maintainers, and promote the co-evolution of production code and test code.

In summary, our work makes the following contributions:

- We conducted an empirical study to investigate the co-evolution between production and test cases in 731 open-source Java projects. Through quantitative analyses, we proved that outdated test case prediction in method granularity is necessary, as well as types of production method changes that are likely to trigger updates to test cases.
- We propose a fine-grained machine learning-based approach, DRIFT, for detecting outdated test cases. DRIFT learns from historical production-test co-evolution by extracting method-level characteristics, and utilizes a more accurate traceability method to identify the test cases that require updates.
- We implemented our approach as a tool and evaluated it using 40 real-world Java projects. Compared to the state of the art, our approach achieved better performance (e.g., a 8.5% improvement in accuracy). The results confirm the effectiveness and potential usefulness of our approach.

The remainder of this paper is organized as follows. Section 2 provides an example to explain why it is necessary to predict test

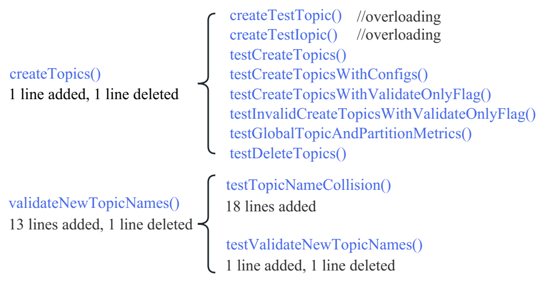
createTopics()
1 line added, 1 line deleted
createTestTopic() //overloading
createTestIopic() //overloading
testCreateTopics()
testCreateTopicsWithConfigs()
testCreateTopicsWithValidateOnlyFlag()
testInvalidCreateTopicsWithValidateOnlyFlag()
testGlobalTopicAndPartitionMetrics()
testDeleteTopics()
validateNewTopicNames()
13 lines added, 1 line deleted
testTopicNameCollision()
18 lines added
testValidateNewTopicNames()
1 line added, 1 line deleted

Figure 1: Production and test code change in Kafka.

updates at the method level. Section 3 presents our empirical study. Section 4 describes our fine-grained machine learning approach for test update prediction. Section 5 presents the evaluation of our approach, followed by a discussion of threats to validity and future work in Section 6. We describe related studies in Section 7 and conclude our work in Section 8.

2 Motivating Example

In the open-source project Apache Kafka [6], a distributed event streaming platform used by thousands of companies, there is a commit (87aa825) made by the developer dengziming on April 14, 2022. The changes to the production code include modifications to the `createTopics` method and the `validateNewTopicNames` method. As shown in Fig. 1, in the test code, the `validateNewTopicNames` method has two related test methods, and the `createTopics` method has eight related test methods. We observed that when the production code changed, only two test methods that test `validateNewTopicNames` production method were updated simultaneously, while the eight test methods related to the `createTopics` production method did not co-evolve.

Observation 1. From the example, we can see that when the production code changes, only a few test methods in the test class need to be updated accordingly. If we predict outdated test code at the class level (i.e., suggesting that the entire test class needs to be updated), developers still need to spend a lot of effort to identify which test methods or test cases need modification. Such coarse-grained predictions can lead to inefficient test maintenance and hinder the practical adoption of the technique.

Observation 2. In addition, we can observe from Fig. 1 that among the eight test methods realted to the `createTopics` method, only the `testCreateTopics` test method adheres to the naming convention. Similarly, for the two test methods that test the `validateNewTopicNames` method, only `testValidateNewTopicNames` conforms to the naming convention. The `testTopicNamesCollision` test method is named in consideration of the specific purpose of the test case. Thus, it is clear that the associations between production and test methods cannot be effectively and reliably established solely by using the naming convention.

Motivated by the above two observations, we propose a new method Drift to predict outdated test cases. In contrast to existing approaches that make predictions at the class level, Drift makes finer-grained predictions at the method level. Rather than relying solely on naming conventions to establish traceability links between production code and test code, Drift takes a holistic approach, considering both improved naming conventions and multi-layer calling relationships between methods, to accurately associate production methods with their related test cases.

3 Empirical Study

As previously mentioned, existing techniques rely on manually identified features and make coarse-grained predictions of outdated test code at the class level. In this section, we perform an empirical study on a large number of open-source Java projects to demonstrate that with class-level predictions, developers still need to expend much effort to identify outdated test methods. Additionally, we aim to automatically identify code-level features to achieve more accurate predictions of outdated test code.

3.1 Data Collection

In order to ensure the validity and generalizability of our empirical findings, it is crucial to collect large-scale, popular, well-maintained, and diversified open-source Java projects to form a dataset for later analysis. To achieve this, we collected projects from GitHub, the world's largest open-source project hosting platform.

Our collection process consists of the following steps. First, we used GitHub's API to collect the URL list of open-source projects. We set the programming language of the project to Java and the number of stars and forks to be greater than 2,000 and 500, respectively, to search for suitable projects. The number of stars and forks represents the popularity of the project. The higher the number of stars and forks, the more popular the project is. With the search, we obtained 1,201 Java projects. Second, we discard projects with no more than 200 commits as such projects are often small-scale. Third, because we focus on the unit testing scenario in this research, we search for `@Test` annotations in the projects to filter out projects without unit tests. Next, we classified projects into the following nine categories according to the main usage of Java in real world: embedded field, big data technology, software tools, web applications, Android system and applications, games, third-party trading systems, server applications in financial industry, and scientific applications. To balance the size of each category, for four categories[1] with an extremely small number of projects, we searched on GitHub again without setting constraints on the number of stars and forks and added the top Java projects (ranked according to the number of stars) with unit tests to supplement them.

Following the above steps, we constructed a dataset consisting of 751 projects that are not only popular, but also well-maintained and diversified. All projects in the dataset have accompanying unit test cases, which makes promoting the co-evolution of production and test code highly necessary. For subsequent experimental evaluations of our prediction tool, we randomly selected 20 projects (the projects will be listed in Section 5) from the dataset. The remaining 731 projects were reserved for our empirical study.

3.2 Research Questions

Our empirical study investigates two research questions:

- **RQ1**: Is it necessary to refine the granularity of outdated test code prediction to the method level?

[1]They are server applications in financial industry, third-party trading systems, embedded field, and scientific applications.

- **RQ2**: What kinds of code changes in production methods would likely cause corresponding test cases to be updated?

3.3 Methodology and Results

To answer the above research questions, we analyzed the historical commits of the 731 open-source Java projects in our dataset. First, we define positive and negative samples for the co-evolution of production code and test code. A positive sample refers to the case where after the production code (class or method) is changed, the corresponding test code (class or test case) is updated within 48 hours. A negative sample refers to the case where after the production code (file or method) is changed, the corresponding test code (class or test case) is not updated within 480 hours.

Associating production and test code. One important step in identifying positive and negative examples is to associate the test code with the corresponding production code. In the following, we present our approach to finding such associations.

A common method to find the traceability link relationship between production file and test file is to use the naming convention [17, 19] (e.g., developers are suggested to add "test" before or after the name of the production file to form the name of the corresponding test file). Through the analysis of 731 open-source projects, we found that the naming convention-based method has two limitations. First, if we use simple rules, we will miss many test files (e.g., some developers add "testCase" to a production file's name to make up the test file's name). In addition, if the two Java files are in different folders, the file names may be the same, which will lead to confusion about traceability link relationships that are found using the naming convention. To address the above limitations, we make improvements when using the naming convention to find corresponding relationships. For files with the same name, we detect modules imported in test files to find corresponding production files. Moreover, when using the naming convention, we check whether the production file's name plus "test", "testCase" or "tests" exists to find the corresponding test file.

When looking for the traceability link relationships between the production methods and test cases, we look for the multi-layer call relationship at the method granularity according to the parsing results of antlr4 grammar parser [13]. The search details of the multi-layer call relationship are as follows. First, we create an empty list for each test case. Then we find the production methods and the other test cases called in each test case and save the called production methods' paths to the empty list. For the called production methods, we continue to search for the other production methods called in them, and save the production methods' paths called. For the called test cases, we repeat to find the call relationships, and further we search for the call relationships of the called production methods and called test cases, respectively. We conduct the above search until the call relationship exceeds the scope of five layers. Finally, we obtain a list of production methods called by each test case in multiple layers, from which we can get the corresponding relationships between production methods and test cases.

3.3.1 RQ1 To investigate RQ1, we find the traceability link relationships between production files and test files in 731 projects. Then we obtain positive and negative samples of file granularity by comparing the change times of the two files. We counted the

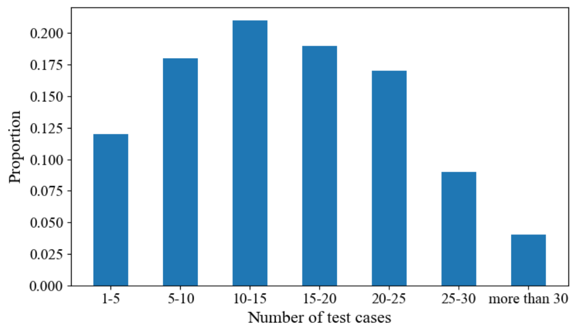

Figure 2: Distribution of the test cases' number in test file

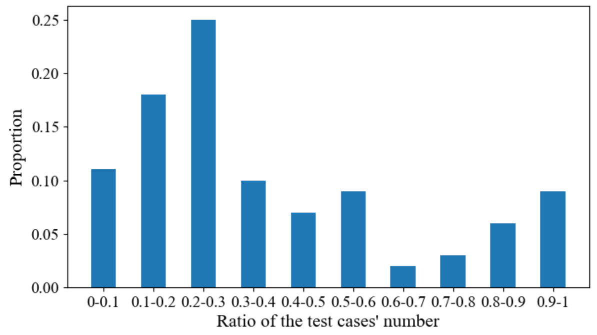

Figure 3: Distribution of the updated test cases' proportion

total number of test cases contained in each positive sample's test file and the number of test cases that were updated in the test file. Fig. 2 gives the distribution of the results for the 731 Java projects. We can see that the number of test cases in the test files is primarily concentrated in the range of 1-35, and the proportion in the interval of 10-15 is the highest, indicating that it is most common that the test files in the positive samples contain 10-15 test cases.

Next, We divided the number of updated test cases in each test file by the total number of test cases in the test file to obtain the proportion of updated test cases. Fig. 3 shows the distribution of such calculated proportions among all positive examples. It can be seen that it is most common that the updated test cases in the test file account for 20% to 30% of the total number of test cases, and it is rare that all test cases in a test file get updated.

In addition, for each test file in positive examples, we counted the number of test cases that test the changed production methods and how many of these test cases were updated along with the production methods. If the former number is not 0 but the latter number is 0 (i.e., none of the test cases for the changed production methods are updated), we classify this situation as "not change". If the former number is not 0 and the latter number is also not 0 (i.e., test cases for the changed production methods are updated), we classify this situation as "should change". For each test file, we calculate the proportion of changed production methods for which the test cases are not updated. Fig. 4 shows the distribution of such proportions among all positive samples. It can be seen from the figure that for many positive samples, at least 30% to 40% of the

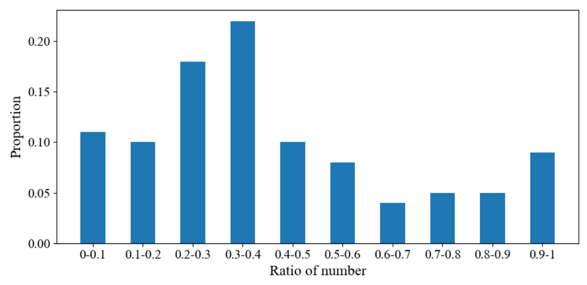

Figure 4: Distribution of changed production method that did not cause test case updates's proportion

Table 1: AST node types defined in Java documentation

AST node types			
Wildcard	LambdaExpression	TypeParameter	Unary
Throw	MethodInvocation	Return	AnnotatedType
Try	CompoundAssignment	Continue	Binary
Break	Erroneous	Block	MemberSelect
DoWhileLoop	ArrayAccess	VariableDeclaration	TypeCast
WhileLoop	ConditionalExpression	ForLoop	Parenthesized
Switch	Annotation	Synchronized	NewArray
EnhancedForLoop	InstanceOf	LabeledStatement	CompilationUnit*
Assert	Identifier	If	NewClass
ExpressionStatement	Assignment	Catch	Literal
ArrayType	IntersectionType	UnionType	PrimitiveType
Modifiers	MethodDeclaration	ParameterizedType	EmptyStatement*
Case	ClassDeclaration*	MemberReference	Import*

production method changes do not trigger test case changes. If a class-level predictor points out that the test class should be updated, developers will still have a hard time identifying which test cases to update.

Finding 1: When a test class is updated due to production code changes, many of its constituent test cases do not need to be updated, even though the corresponding production methods are changed. Refining the prediction granularity of outdated test code to the method level can significantly reduce the effort of identifying outdated test cases in test files.

3.3.2 RQ2 In order to find out what kinds of code changes in production methods would likely cause test cases to be updated, we compared the ASTs (Abstract Syntax Trees) before and after production method updates and performed further analysis. We first extracted all 52 AST node types from the Java Development Kit's `Tree.java` file, as shown in Table 1, and these AST node types will be used to describe code changes.

We simply screen the above AST node types using the basic VarianceThreshold method [3], which will delete features whose variance does not meet a certain threshold. The specific variance calculation formula is:

$$Var[X] = p(1 - p)$$

X represents the node type we are screening, and p represents the proportion of samples containing this node type in all samples of the above dataset.

Since the greater the variance of a variable, the larger its contribution and effect to the model will be, the node type with a large variance should be retained, and the other meaningless variables should be eliminated. We set the screening threshold to 0. The purpose of this setting is to roughly screen out node types that exist or do not exist in both positive and negative samples of the method granularity. After screening, we extracted 48 AST node types (those not annotated with "*" in Table 1) as production code change types that may be related to co-evolution. These node types will be considered as features in designing our machine learning-based technique for predicting outdated test cases.

Finding 2: 48 production code change types in Table 1 would likely cause the corresponding test code to be updated.

4 The Drift Approach

The high-level workflow of Drift is presented in Fig. 5. The method is divided into three parts: (1) data preprocessing, (2) method-granularity feature extraction, and (3) classifier training. The specifics of each part are described in the following sections.

4.1 Data Preprocessing

The goal of data preprocessing is to identify positive and negative samples of method granularity from historical commits of software projects. In the case of an open-source Java project, the following data preprocessing operations are performed: First, we extract all changed production file names from historical commits and collect all file names that have existed in the project. Second, we look for the traceability link relationship between the changed production file and test file. Third, we identify all production methods that have changed in the production file and find the traceability link relationship between the changed production methods and test cases. For each production method change, we check if the corresponding test cases in the historical commits have been updated. If the corresponding test cases have never been updated, we store the production method change and the corresponding test cases' locations together as a negative sample, labeled as NC (not changed). If the corresponding test cases have been updated, we find the test case update closest to the production method change and calculate the interval time between the two changes. If the shortest interval time is less than 48 hours, and between the two changes, the production method has not changed again, we store the production method change and the corresponding test case updates together as a positive sample, label it as SC (should change). If the shortest interval time is greater than 480 hours, the production method change and the locations of the corresponding test cases will also be stored as a negative sample. The specific flow chart for the data preprocessing is presented in Fig. 6. When looking for the traceability link relationship between the changed production file and test file, as well as the traceability link relationship between the production method and test cases, we follow the process in Section 3.3.

4.2 Method-Granularity Feature Extraction

After the positive and negative samples of co-evolution of method granularity are obtained through data preprocessing, we extract

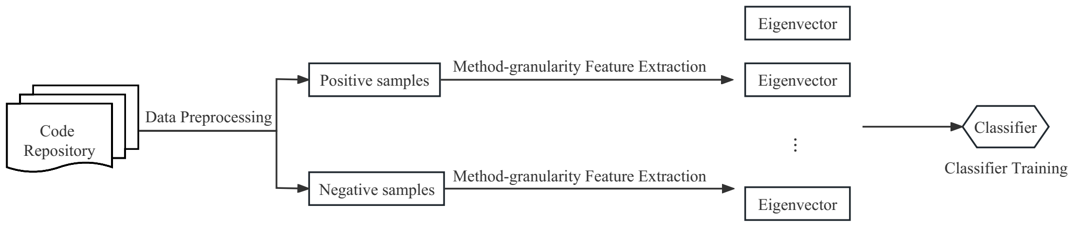

Figure 5: High-level workflow of DRIFT

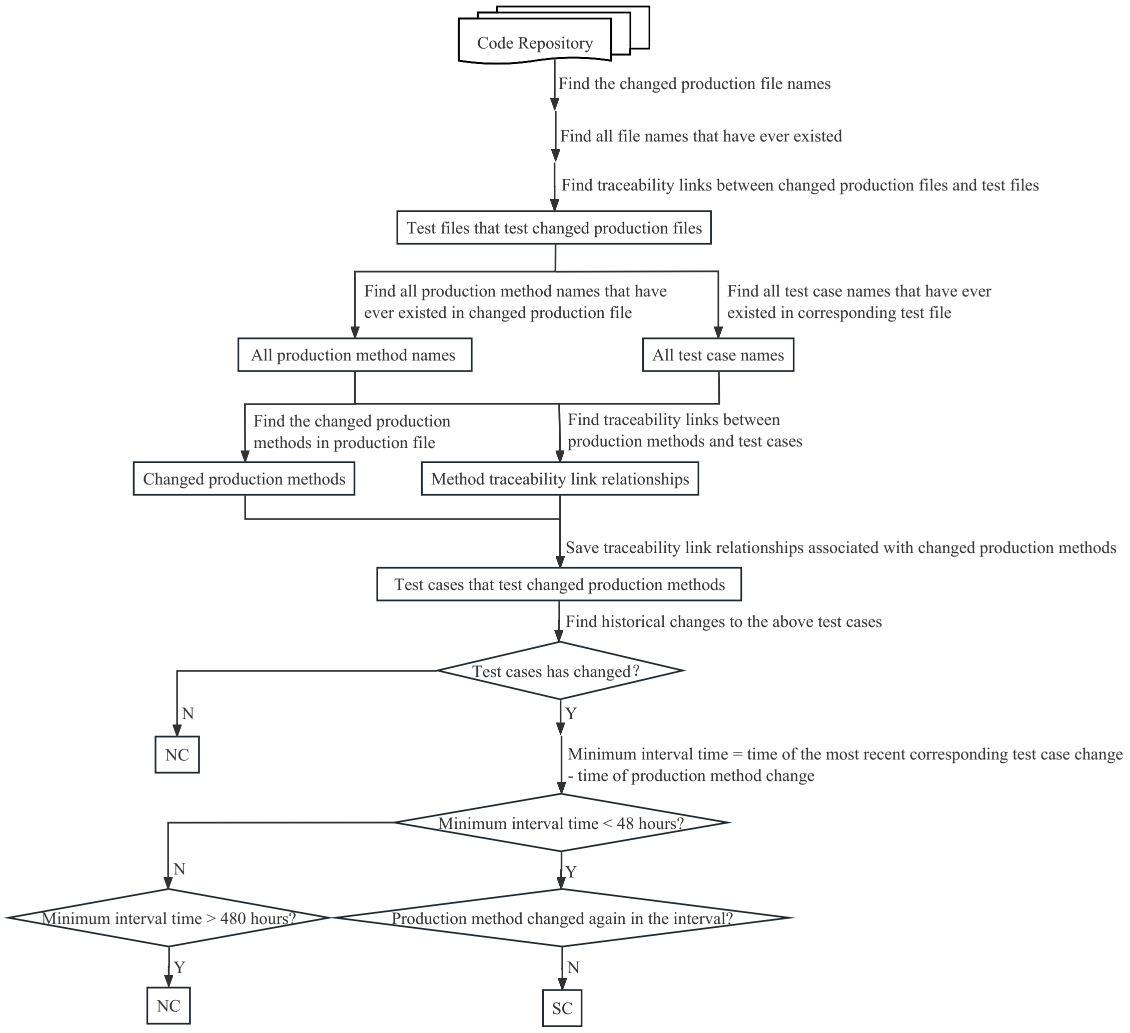
Code Repository
Find the changed production file names
Find all file names that have ever existed
Find traceability links between changed production files and test files
Test files that test changed production files
Find all production method names that have ever existed in changed production file
Find all test case names that have ever existed in corresponding test file
All production method names
All test case names
Find the changed production methods in production file
Find traceability links between production methods and test cases
Changed production methods
Method traceability link relationships
Save traceability link relationships associated with changed production methods
Test cases that test changed production methods
Find historical changes to the above test cases
Test cases has changed?
N
NC
Y
Minimum interval time = time of the most recent corresponding test case change - time of production method change
Minimum interval time < 48 hours?
N
Y
Minimum interval time > 480 hours?
Production method changed again in the interval?
Y
NC
N
SC

Figure 6: Detailed steps of data preprocessing

the features of method granularity for the positive and negative samples, and represent them by eigenvectors. The features used in this step are from the 48 production code change types (AST node types) listed in Table 1 that may result in test case updates (each AST node type represents a structure in the source code). The 48 features that might lead to coevolution have been initially screened to exclude node types that are completely unrelated to coevolution.

Since each production code change type in the Java file can correspond to an interval, we parse the changed production code to identify the specific line number range of each production code change type (listed in Table 1) in the code. Then we judge whether the changed code lines in each production method are within the range of production code change types by code line unit. Since there are two types of operations for code lines, namely, adding and deleting, each feature is represented by two dimensions of the feature vector, one representing the number of lines added by the feature in the sample, and the other representing the number of lines deleted by the feature in the sample.

4.3 Classifier Training

After extracting eigenvectors of positive and negative samples at method granularity, we use a machine learning classifier to learn the rules contained in the eigenvectors and train the classifier to predict the correct label of the input. It's important to consider the balance of positive and negative samples during classifier training. If the difference in the number of positive and negative samples is too large, it can lead to overfitting of the classifier and poor generalization ability of the trained model. To address this, we undersample the class with too many samples to balance.

During classifier training, we input the eigenvectors extracted from the samples into the classifier. The output of the classifier is the category (label) of the samples (NC or SC). The classifier learns how to map the input data (eigenvectors) to the correct category (label) during the training process. This enables the classifier to predict whether test cases need to be updated for a certain production method change, ultimately saving time and resources for the organization. We compare the classification performance of six commonly used machine learning algorithms: Support Vector Machines, Random Forests, Naive Bayes, K-Nearest Neighbor, Gradient Boosting, and Logistic Regression. Our aim is to determine the best classifier algorithm for DRIFT.

To collect positive and negative samples from 40 projects, we refer to the data preprocessing of DRIFTin Section 4.1. We use standard classifiers from the `scikit-learn` library and set default classifier hyperparameters. We train six classifier models and use the ten-fold cross-validation method to obtain average precision and recall, then draw the PR curve (Precision-Recall curve).

5 Evaluation

As the choice of classifier in the DRIFT method can significantly affect the prediction performance of outdated test cases, it is important to compare the performance of various commonly used classifiers to identify the most suitable algorithm for the DRIFT method. Then we evaluate the prediction performance of DRIFT in within-project and cross-project scenarios, and compare it with the rule-based outdated test case prediction method [10] and the SITAR [19]. The within-project scenario involves learning the co-evolution patterns from the historical commits of a given project and predicting the outdated test cases of that same project. The cross-project scenario, on the other hand, involves learning the co-evolution patterns from historical commits of other projects and predicting the outdated test cases of a new project. So we evaluate DRIFT based on the following four research questions:

- **RQ3**: What kind of classifier works best in the scenario of outdated test case prediction?
- **RQ4**: What is the performance of DRIFT in within-project and cross-project scenarios?
- **RQ5**: How does DRIFT compare to rule-based baselines in terms of prediction performance?

Table 2: List of projects for evaluation

Projects	# stars	# forks	# commits	LoC
macrozheng/mall	64.3k	26.4k	937	181,063
elastic/elasticsearch	63.1k	22.9k	67,937	4301,233
ReactiveX/RxJava	46.9k	7.7k	6,028	485,496
Blankj/AndroidUtilCode	32k	10.6k	1,430	136,580
zxing/zxing	30.8k	9.3k	3,684	219,738
alibaba/fastjson	25.2k	6.5k	3,980	278,505
libgdx/libgdx	21.3k	6.4k	15,205	1075,483
jenkinsci/jenkins	20.5k	8k	33,472	736,128
perwendel/spark	9.5k	1.6k	1,067	22,224
hs-web/hsweb-framework	8k	3k	2,798	45,905
apache/hbase	4.8k	3.2k	19,642	1477,258
pardom-zz/ActiveAndroid	4.7k	1.1k	284	6,380
MovingBlocks/Terasology	3.5k	1.3k	11,916	386,859
kiegroup/optaplanner	3k	902	8,808	4803,616
apache/opennlp	1.2k	420	2,030	211,422
ahmetaa/zemberek-nlp	1k	204	1,358	598,048
OpenGamma/Strata	734	258	4,645	860,971
DSpace/DSpace	718	1.2k	16,570	630,721
jnode/jnode	313	116	6,169	23,907
bitpay/java-bitpay-client	34	59	383	168,734
apache/activeMQ	2.1k	1.4k	11,365	1019,335
apache/cloudStack	1.4k	989	35,756	1925,030
apache/commons-math	490	340	7,127	311,381
apache/flink	20.9k	11.8k	33,052	3298,122
apache/geode	2.2k	680	11,248	2287,249
apache/james-project	683	422	14,443	1206,402
apache/logging-log4j2	3k	1.5k	12,659	464,672
apache/storm	6.4k	4.1k	10,537	749,335
apache/usergrid	995	435	10,954	1005,374
apache/zeppelin	6k	2.7k	5,228	490,960
SERG-Delft/jpacman-framework	113	249	374	7,185
google/gson	22k	4.2k	1,803	50,073
pmd/pmd	4.2k	1.4k	25,875	583,204
biojava/biojava	523	362	6,707	1703,879
izpack/izpack	300	263	5,713	255,498
JodaOrg/joda-time	4.9k	953	2,263	160,848
dnsjava/dnsjava	769	208	2,084	90,836
FasterXML/jackson-databind	3.2k	1.3k	6,902	284,947
jruby/jruby	3.7k	925	52,230	1141,798
jhy/jsoup	10k	2.1k	1,785	39,590

- **RQ6**: How does DRIFT compare to the state-of-the-art approach SITAR in terms of prediction performance?

We randomly selected 20 projects from the dataset collected in Section 3.1 for evaluation. In addition, we added the 20 projects used to evaluate SITAR's in the exsiting work [19] to form a dataset containing 40 open-source Java projects. Table 2 lists the projects.

5.1 RQ3: Comparison of Classifiers

5.1.1 Experimental Setup We compare the classification performance of six commonly used machine learning classifier algorithms, including support vector machines, random forest, naive Bayes, K-nearest neighbor, gradient boosting, and logistic regression, then find out the best classifier algorithm in this scenario. First,we collect the positive and negative samples of method granularity from the historical submission records of each project. For specific collection steps, please refer to the data preprocessing part of DRIFT in Section 4.1. The classifiers we use are all standard classifiers in the `scikit-learn` library, and we use the default classifier hyperparameters from the `scikit-learn` library. Refering to the method-granularity feature extraction and classifier training of DRIFT in Section 4, we perform feature extraction on the positive and negative samples of method granularity, and train six classifier models. We use the ten-fold cross-validation method to obtain the average

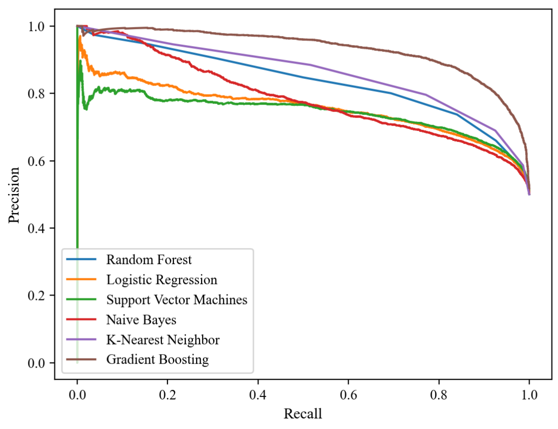

Figure 7: PR curves for six machine learning classifiers

precision and average recall, and draw the PR curve (Precision-Recall curve).

5.1.2 Results In the figure shown (Fig. 7), it is evident that when predicting unit test case updates for changed production methods using Drift, the PR curve of the gradient boosting surpasses that of the other five commonly used classifier algorithms. Compared to the other algorithms, gradient boosting consistently achieves higher precision at fixed recall. While K-nearest neighbor and random forest are slightly inferior to gradient boosting, naive Bayes shows a significant decline in performance when the recall rate exceeds 0.1. For support vector machines and logistic regression, when the recall approaches 0, the PR curves exhibit a sharp decline due to many false positives of outdated test cases. In conclusion, we have chosen gradient boosting as the classifier algorithm in Drift.

5.2 RQ4: Performance of Drift

5.2.1 Experimental Setup In the within-project scenario, as described in Section 4, we searched for positive and negative samples of method granularity in the 40 projects listed in Table 2, and then we performed feature extraction and trained a gradient boosting classifier model using the ten-fold cross-validation method. We averaged the prediction results from ten runs to obtain the average accuracy, precision, and recall. In the cross-project scenario, we obtained 731 open-source Java projects from Section 3.1, and searched for positive and negative samples of method granularity using the process outlined in Section 4.1. After feature extraction, we finally trained a gradient boosting classifier model. Then we tested the trained classifier using the samples extracted from the 40 projects listed in Table 2. We ran the test ten times independently and calculated the average values as the final result.

5.2.2 Results Table 3 shows the prediction performance of 40 projects using Drift for outdated test case prediction in within-project and cross-project scenarios. In order to save space, we simplify the projects' names. In within-project scenario, the accuracy is between 67.8% and 95.9%, with an average of 81.6%. The F1 is between 64.7% and 92.7%, and the average F1 is 80.6%. Although

Table 3: Performance of Drift

Projects	within-project				cross-project			
	Acc.	Pre.	Rec.	F1	Acc.	Pre.	Rec.	F1
mall	78.9	82.0	77.6	79.7	71.7	70.8	70.1	70.4
elasticsearch	77.1	76.1	75.2	75.6	68.1	69.8	66.2	68.0
RxJava	76.7	76.2	73.9	75.0	63.2	65.9	62.1	63.9
AndroidUtilCode	72.3	74.5	70.1	72.2	67.8	69.1	63.5	66.2
zxing	77.8	80.4	72.9	76.5	71.4	71.6	65.2	68.3
fastjson	76.5	78.2	71.5	74.7	68.2	70.5	62.3	66.1
libgdx	87.2	87.9	83.0	85.4	72.6	73.0	71.9	72.4
jenkins	70.1	71.6	69.5	70.5	67.2	67.5	65.8	66.6
spark	78.5	77.6	70.1	73.7	71.6	61.5	71.3	66.0
hsweb	81.2	83.9	76.0	79.8	72.2	73.6	70.1	71.8
hbase	77.0	79.4	75.0	77.1	60.4	64.7	62.5	63.6
ActiveAndroid	72.4	76.8	67.8	72.0	69.2	72.8	59.8	65.7
Terasology	73.5	75.4	70.8	73.0	71.4	73.8	66.7	70.1
optaplanner	82.3	85.6	78.1	81.7	74.6	73.1	71.6	72.3
opennlp	74.2	78.1	71.3	74.5	55.1	56.3	52.8	54.5
zemberek-nlp	67.8	68.4	61.4	64.7	60.4	67.2	60.5	63.7
Strata	76.9	79.5	73.1	76.2	67.5	73.1	61.3	66.7
DSpace	73.2	72.9	62.0	67.0	52.9	72.0	76.5	74.2
jnode	75.8	77.0	69.7	73.2	62.0	61.7	67.5	64.5
bitpay	72.1	72.6	70.4	71.5	64.8	77.8	77.6	77.7
activeMQ	85.2	83.9	81.3	82.6	70.5	74.7	68.1	71.2
cloudStack	91.1	93.2	82.5	87.5	82.5	80.1	71.8	75.7
math	84.5	85.2	83.6	84.4	69.7	71.2	66.9	69.0
flink	90.1	91.2	85.8	88.4	79.2	80.1	78.4	79.2
geode	90.3	92.6	85.1	88.7	71.8	71.4	71.2	71.3
james	83.9	88.5	83.1	85.7	78.5	79.2	76.5	77.8
logging-log4j2	84.8	89.7	81.4	85.3	69.2	71.6	69.0	70.3
storm	95.9	96.3	89.4	92.7	70.1	70.2	69.5	69.8
usergrid	95.1	94.8	89.5	92.1	82.5	82.6	81.5	82.0
zeppelin	85.5	88.2	85.9	87.0	64.5	65.2	64.6	64.9
jpacman	91.6	96.7	88.0	92.1	81.0	85.4	80.0	82.6
gson	83.7	85.2	84.1	84.6	72.4	73.2	70.8	72.0
pmd	84.6	87.5	78.9	83.0	72.5	76.4	71.8	74.0
biojava	91.8	92.4	85.2	88.7	62.3	63.6	61.0	62.3
izpack	82.2	83.4	81.1	82.2	60.2	63.1	65.3	64.2
joda-time	84.3	85.7	80.3	82.9	75.8	77.4	69.9	73.5
dnsjava	93.5	94.2	86.5	90.2	65.4	67.5	63.1	65.2
jackson	83.5	85.2	79.6	82.3	71.2	73.9	69.0	71.4
jruby	80.6	82.3	80.9	81.6	61.8	69.0	61.7	65.1
jsoup	87.0	89.2	87.8	88.5	65.0	67.8	60.5	63.9
Avg.	81.6	83.4	77.9	80.6	68.9	71.2	67.9	69.5

the above metrics fluctuate with different projects, they generally perform well. In the `apache/storm` project, the accuracy can even reach 95.9%, and the F1 is 92.7%. This shows that in real-world Java projects, Drift can effectively identify outdated test cases. In the cross-project scenario, the accuracy of Drift ranges from 55.1% to 82.5%, with an average of 68.9%. The range of F1 is 54.5% to 82.6%, with an average of 69.5%. Although the above metrics are lower than those in within-project scenarios, they perform well in some projects. For example, in `apache/usergrid` and `SERG-Delft/jpacman-framework`, all values are greater than 80%. It shows that Drift can also have satisfactory performance in the scenario of cross-project, and the co-evolution laws learned from 731 open-source projects can guide the prediction of outdated test cases in new projects.

5.3 RQ5: Comparison with Rule-based Baselines

5.3.1 Experimental Setup The three co-evolution patterns mined by ChangeDistiller [10] are: Creation/deletion of a production class results in creation/deletion of a corresponding test class (R1); Creation/modification of a production method causes corresponding test cases to be updated (R2); Changes in conditional statements in production code result in updated test cases (R3). Since R1 focuses on the class level, we transform R1 into: when creating/deleting a production method, if other methods in the production class are all created/deleted, the test cases corresponding to the production method should be created/deleted.

We use the above three patterns for outdated test case prediction respectively. The data for our experiment comes from 40 projects in Table 2. We use the samples of method granularity in each project to test the prediction performance of each co-evolution pattern. In addition, we combine three patterns for outdated test case prediction. We independently test ten times and obtain the average value of the results to represent the final performance.

5.3.2 Results The prediction performance of outdated test cases based on three co-evolution patterns is shown in Table 4. To save space, we simplify the projects' names. R1 stipulates that when a production class is created or deleted, the corresponding test class is also created or deleted. However, this situation does not occur frequently, so the average recall is only 30.1% when R1 is used for outdated test case prediction, and the average F1 is 44.8%. In R2, the creation/modification of production methods appears more frequently, so the average recall reaches 86.2%, but because the co-evolution pattern described by R2 is not in line with reality, the precision is only 52.8%. The precision and recall of R3 are relatively balanced, but compared with Drift, the performance is not outstanding. When combining R1, R2 and R3, the performance is still poor compared to Drift, indicating that the combination of the above three co-evolution patterns cannot effectively describe the co-evolution situation. Therefore, Drift predicts outdated test cases better than rule-based outdated test case prediction methods.

Table 4: Performance of rule-based prediction methods

Projects	R1			R2			R3			R1 & R2 & R3		
	Pre.	Rec.	F1	Pre.	Rec.	F1	Pre.	Rec.	F1	Pre.	Rec.	F1
mall	97.6	24.6	39.3	53.7	78.8	63.9	55.4	39.5	46.1	52.0	86.8	65.0
elasticsearch	90.1	14.2	24.5	51.6	96.7	67.3	70.3	64.2	67.1	51.6	97.9	67.6
RxJava	99.6	30.2	46.3	55.6	89.7	68.6	72.2	42.9	53.8	54.5	92.2	68.5
AndroidUtilCode	98.2	24.8	39.6	51.9	87.1	65.0	74.2	48.4	58.6	41.3	89.1	56.4
zxing	98.3	19.9	33.1	55.9	88.4	68.5	65.1	44.6	52.9	54.3	91.5	68.2
fastjson	95.2	38.9	55.2	52.6	90.7	66.6	73.8	70.4	72.1	51.5	92.5	66.2
libgdx	96.7	42.2	58.8	49.7	76.1	60.1	68.2	53.0	59.6	55.1	82.9	66.2
jenkins	98.9	15.9	27.4	51.1	94.4	66.3	69.3	48.8	57.3	50.8	96.3	66.5
spark	90.7	43.2	58.5	59.7	76.1	66.9	68.2	53.1	59.7	50.5	82.9	62.8
hsweb	95.9	15.9	27.3	51.1	93.4	66.1	69.3	48.8	57.3	34.8	94.3	50.8
hbase	98.3	19.9	33.1	55.9	88.4	68.5	65.1	44.6	52.9	54.3	91.5	68.2
ActiveAndroid	99.7	38.9	56.0	52.6	90.7	66.6	73.8	70.4	72.1	51.5	92.5	66.2
Terasology	96.7	42.2	58.8	49.7	76.1	60.1	68.2	53.6	60.0	50.5	82.9	62.8
optaplanner	97.4	34.5	51.0	42.0	77.7	54.5	54.2	48.0	50.9	55.2	79.4	65.1
opennlp	98.5	38.9	55.8	52.6	90.7	66.6	73.8	70.4	72.1	51.5	92.2	66.1
zemberek-nlp	91.8	42.3	57.9	57.9	77.6	66.3	68.2	54.5	60.6	51.3	82.9	63.4
Strata	99.7	39.7	56.8	56.1	93.2	70.0	72.3	69.9	71.1	50.1	90.6	64.5
DSpace	96.7	42.2	58.8	49.7	76.1	60.1	68.2	53.4	59.9	50.5	82.4	62.6
jnode	98.5	29.4	45.3	45.7	80.9	58.4	57.4	40.2	47.3	49.6	87.1	63.2
bitpay	92.1	14.2	24.6	51.6	96.7	67.3	70.3	64.2	67.1	51.6	97.9	67.6
activeMQ	98.6	30.2	46.2	55.6	89.7	68.6	72.2	42.9	53.8	54.5	92.2	68.5
cloudStack	99.2	24.8	39.7	52.0	87.1	65.1	74.2	48.4	58.6	51.2	89.1	65.0
math	98.3	19.9	33.1	55.9	88.4	68.5	65.1	44.6	52.9	54.3	91.5	68.2
flink	96.7	39.9	56.5	53.6	88.7	66.8	74.8	71.5	73.1	52.6	91.8	66.9
geode	94.3	41.2	57.3	52.7	75.9	62.2	69.3	51.4	59.0	52.2	82.9	64.1
james	98.9	15.9	27.4	51.1	94.4	66.3	69.3	48.8	57.3	50.8	96.3	66.5
logging-log4j2	91.4	39.8	55.5	60.1	75.3	66.8	70.7	52.9	60.5	48.4	82.8	61.1
storm	90.7	43.2	58.5	59.7	76.1	66.9	68.2	53.1	59.7	50.5	82.4	62.6
usergrid	95.9	15.9	27.3	51.1	93.4	66.1	69.3	48.8	57.3	34.8	94.3	50.8
zeppelin	98.3	19.9	33.1	55.9	88.4	68.5	65.1	44.6	52.9	54.3	91.0	68.0
jpacman	99.7	38.9	56.0	52.6	90.7	66.6	73.8	70.4	72.1	51.5	92.8	66.2
gson	97.8	43.1	59.8	51.9	74.8	61.3	70.1	49.4	58.0	48.2	84.6	61.4
pmd	98.9	15.9	27.4	51.1	94.4	66.3	69.3	48.8	57.3	50.8	96.3	66.5
biojava	94.5	38.2	54.4	49.8	80.5	61.5	62.9	54.2	58.2	49.9	86.3	63.2
izpack	99.5	33.8	50.5	51.6	86.0	64.5	77.3	55.2	64.4	51.9	89.7	65.8
joda-time	95.9	15.9	27.3	51.1	93.4	66.1	69.3	48.8	57.3	34.8	94.3	50.8
dnsjava	98.3	19.9	33.1	55.9	88.4	68.5	65.1	44.6	52.9	54.3	89.3	67.5
jackson	99.7	38.9	56.0	52.6	90.7	66.6	73.8	70.4	72.1	51.5	91.6	65.9
jruby	96.7	41.3	57.9	49.7	79.2	61.1	68.7	57.5	62.6	50.5	81.7	62.4
jsoup	95.9	15.9	27.3	51.1	93.4	66.1	69.3	48.8	57.3	34.8	94.6	50.9
Avg.	96.7	30.1	44.8	52.8	86.2	65.3	68.9	53.5	59.9	49.9	89.5	63.8

5.4 RQ6: Comparison with Sitar

5.4.1 Experimental Setup Due to the different prediction granularity of Drift and Sitar, it is difficult to use the same set of samples to compare their prediction performance, so we deal with the experimental data and results as follows. First, we use the improved naming convention in Section 3.3 to determine the traceability link relationship at file level in 40 open-source projects, and find all samples at file level based on the change times of files. Then we experiment with Sitar using the ten-fold cross-validation method to get the average prediction performance. When experimenting with Drift, we use the ten-fold cross-validation method to divide all positive and negative samples at the file level into ten, and collect the samples at the method level contained in each piece of data. The collection steps of samples at the method level refer to Section 4.1. We put the samples at the method level in any nine pieces of data into the classifier to train Drift, and use the remaining one piece of data to test the trained classifier. Then we merge the prediction results of samples if the test cases of samples are in the same test file. If the prediction results of each sample are not changed, the corresponding test file is predicted not to change. If there are samples that are predicted to update, we predict that the corresponding test file should change. We conduct experiments using ten-fold cross-validation and obtain the average prediction performance.

5.4.2 Results The prediction performance of Sitar and Drift is shown in Table 5. Although the results of Drift are simplified and combined, it still have a higher prediction accuracy than Sitar. On average, Drift has an accuracy of 81.8%, compared to the 73.3% average prediction accuracy of Sitar. Additionally, the average precision and recall of Drift are also higher than those of Sitar. The average F1 score of Drift is 8.3% higher than that of Sitar. For `apache/storm` and `SERG-Delft/jpacman-framework`, the prediction results of Sitar are good, but Drift can have better prediction performance on them, which shows that compared with Sitar, Drift can better learn the co-evolution laws of production code and test code in the historical commits, and Drift has a better prediction performance on outdated test cases.

In addition, since Sitar predicts outdated test code at file granularity while Drift predicts at method granularity, Drift can accurately locate the test cases that need to be updated. From Fig. 3, it can be seen that the most common case is that the number of test cases changed in the test file accounts for 20% to 30% of the total number of test cases in the test file. Therefore, if we assume that Sitar and Drift can both accurately predict test files (or test cases in the test files) that need to be updated, Drift can reduce testers' workload of checking test cases by about 75% compared to Sitar. So Drift can better facilitate the maintenance of test cases, promoting the co-evolution of test code and production code.

6 Discussions

Threats to Validity. Due to the limited number of projects in empirical research, the laws discovered by empirical research may not

Table 5: Performance of Sitar and Drift in file level

Projects	Sitar				Drift			
	Acc.	Pre.	Rec.	F1	Acc.	Pre.	Rec.	F1
mall	70.5	71.7	70.8	71.2	79.3	81.9	78.1	80.0
elasticsearch	72.5	73.1	72.8	72.9	76.5	75.2	75.8	75.5
RxJava	69.7	71.2	70.9	71.0	76.8	76.1	74.2	75.1
AndroidUtilCode	61.2	61.5	61.1	61.3	72.5	73.9	70.4	72.1
zxing	70.8	72.4	65.2	68.6	77.9	79.8	73.7	76.6
fastjson	71.4	73.3	70.2	71.7	76.8	77.1	72.4	74.7
llibgdx	74.6	76.9	72.3	74.5	87.1	87.6	85.4	86.5
jenkins	65.3	69.2	62.5	65.7	70.2	71.5	70.8	71.1
spark	70.1	71.3	66.9	69.0	78.2	77.2	70.6	73.8
hsweb	76.5	78.2	74.6	76.4	81.5	83.2	78.6	80.8
hbase	68.4	69.2	70.8	70.0	77.4	78.9	76.8	77.8
ActiveAndroid	63.0	64.4	50.0	56.3	72.1	76.4	69.7	72.9
Terasology	69.8	71.1	69.7	70.4	73.2	74.5	71.4	72.9
optaplanner	70.2	70.9	69.3	70.1	82.6	85.1	80.5	82.7
opennlp	68.2	72.1	71.8	71.9	74.7	77.9	72.4	75.0
zemberek-nlp	59.8	61.4	55.2	58.1	67.6	68.3	62.8	65.4
Strata	70.9	71.5	68.1	69.8	77.5	79.1	78.0	78.5
DSpace	60.2	65.9	58.6	62.0	73.1	72.3	65.9	69.0
jnode	69.8	72.5	71.7	72.1	75.2	76.1	71.8	73.9
bitpay	55.2	59.6	52.4	55.8	73.2	72.5	70.9	71.7
activeMQ	74.3	76.2	71.4	73.7	85.2	83.1	82.2	82.6
cloudStack	85.3	86.7	80.1	83.3	91.5	92.3	89.4	90.8
math	74.6	73.2	71.4	72.3	84.3	83.6	85.2	84.4
flink	84.9	85.1	85.3	85.2	89.5	88.1	87.5	87.8
geode	82.7	84.9	80.7	82.7	90.8	90.2	89.7	89.9
james	75.5	76.6	72.9	74.7	83.6	87.0	85.3	86.1
logging-log4j2	77.5	80.2	73.5	76.7	84.2	89.4	85.1	87.2
storm	91.0	94.4	90.0	92.1	95.4	97.6	91.3	94.3
usergrid	87.4	89.2	83.0	86.0	95.8	94.9	92.8	93.8
zeppelin	72.5	73.4	71.8	72.6	85.6	87.0	86.1	86.5
jpacman	88.3	94.6	82.8	88.3	91.0	96.4	91.8	94.0
gson	71.9	73.1	71.1	72.1	84.6	84.5	84.6	84.5
pmd	75.8	75.4	74.3	74.8	84.9	86.4	85.7	86.0
biojava	86.9	87.5	81.4	84.3	92.1	92.5	87.6	90.0
izpack	72.2	72.9	73.2	73.0	81.5	81.6	82.2	81.9
joda-time	71.8	72.1	72.0	72.0	82.3	82.5	82.7	82.6
dnsjava	85.0	82.8	89.7	86.1	93.7	93.9	95.2	94.5
jackson	70.5	70.5	77.8	74.0	84.3	84.7	88.3	86.5
jruby	74.9	75.2	72.1	73.6	80.9	81.4	81.1	81.2
jsoup	71.6	71.3	73.1	72.2	87.6	87.5	88.9	88.2
Avg.	73.3	74.8	71.8	73.2	81.8	82.7	80.0	81.5

generally guide the co-evolution of codes. In order to enhance the universality of the rules found in the empirical research, we screen the open-source Java projects with high popularity, then classify and expand the data set according to the usage of Java projects. Since the popularity of projects means the projects are widely used, and a sufficient number of projects with different usages can ensure a more comprehensive data set, the threat is mitigated to some extent. In order to eliminate the deviation of the prediction performance of the classifier and evaluate the generalization ability of the classifier, we use the ten-fold cross-validation method to average the experimental results of the classifier to ensure the performance and effect of the classifier in practical applications.

Future Work. In the future, we will improve the performance of Drift in cross-project scenario to help Drift achieve good performance on poorly maintained projects. In addition, we will extend Drift to other programming languages to expand the application of Drift, so that it can be more widely used in the software product development process in the real world.

7 Related Work

The co-evolution of production code and test code is crucial for ensuring the quality of production code. Many researchers have focused on mining the laws of co-evolution to guide the update of test cases. Zaidman et al. [21] explored the co-evolution law of test code through change history, growth history, and test quality evolution view but did not clearly summarize the co-evolution of production code and test code. Pinto et al. [14] categorized the reasons behind the addition, removal, and modification of test code, while Marsavina et al. [9] studied the co-evolution between production code and test code by constructing the Covrig framework and found some behavioral rules of co-evolution between production code and test code. Levin et al. [7] focused on the co-evolution of semantic changes in 61 popular open-source projects and found co-evolution caused by specific maintenance activities. Wang et al. proposed Sitar [19], which learned the importance of features for co-evolution through a machine learning model and used a classifier to predict outdated test cases. Shimmi et al. [16] proposed the TSE method, which leverages the co-evolution laws of production code and test code to generate and recommend test cases or supplement incomplete test suites with missing test cases.

The most widely used strategy for finding traceability links in existing work [16, 17, 19] is the naming convention (i.e., adding "test" to the name of the unit under test to form the name of the test code). However, White et al. [20] showed that naming conventions can have high accuracy and recall when finding link relationships at the file level, but they are not effective at the method level. In actual project development, using the naming convention to find the link relationship may miss the link relationship between the production code and the test code. In addition, StaticCallGraph [2] and LCBA (Last Call Before Assert) [18] also introduce a lot of noise into the results due to their wide search range.

8 Conclusion

In this work, we conducted a large-scale empirical study based on 731 open-source Java software projects to study the co-evolution of production and test code. We found that when predicting outdated test code, refining the prediction granularity to the method level can generally reduce the workload of test case maintenance. We also observed 48 types of production code changes that may be related to test cases updates.

Based our empirical findings, we propose the Drift method for fine-grained prediction of outdated test cases. We evaluated Drift on 40 open-source Java projects, and found that Drift can have good prediction results in both within-project and cross-project scenarios, which shows that it can learn the co-evolution rules in different projects and use the rules to predict outdated test cases. In addition, the performance of Drift is significantly improved compared to the existing rule-based methods. Compared with the state-of-the-art approach Sitar, the average accuracy of Drift has increased by about 8.5%, the F1-score has increased by about 8.3%, and the number of test cases that testers need to check has decreased by about 75%. These results confirm that Drift can predict outdated test cases in a better and finer-grained manner, thus better promoting the co-evolution of production and test code.

Acknowledgments

This work is supported by the National Natural Science Foundation of China (Grant No. 61932021) and the Guangdong Basic and Applied Basic Research Fund (Grant No. 2021A1515011562).

References

[1] Apache. [n. d.]. *Maven - introduction to the standard directory layout.* http://maven.apache.org/guides/introduction/

[2] Moritz Beller, Georgios Gousios, and Andy Zaidman. 2017. Oops, my tests broke the build: An explorative analysis of travis ci with github. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR).* IEEE, 356–367.

[3] Jason Brownlee. 2020. *Data preparation for machine learning: data cleaning, feature selection, and data transforms in Python.* Machine Learning Mastery.

[4] Zebao Gao, Zhenyu Chen, Yunxiao Zou, and Atif M Memon. 2015. SITAR: GUI test script repair. *Ieee transactions on software engineering* 42, 2 (2015), 170–186.

[5] Mouna Hammoudi, Gregg Rothermel, and Andrea Stocco. 2016. Waterfall: An incremental approach for repairing record-replay tests of web applications. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering.* 751–762.

[6] Jay Kreps. [n. d.]. *Kafka.* https://rdicosmo.github.io/kafka/

[7] Stanislav Levin and Amiram Yehudai. 2017. The co-evolution of test maintenance and code maintenance through the lens of fine-grained semantic changes. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME).* IEEE, 35–46.

[8] Xiangyu Li, Marcelo d'Amorim, and Alessandro Orso. 2019. Intent-preserving test repair. In *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST).* IEEE, 217–227.

[9] Paul Marinescu, Petr Hosek, and Cristian Cadar. 2014. Covrig: A framework for the analysis of code, test, and coverage evolution in real software. In *Proceedings of the 2014 international symposium on software testing and analysis.* 93–104.

[10] Cosmin Marsavina, Daniele Romano, and Andy Zaidman. 2014. Studying fine-grained co-evolution patterns of production and test code. In *2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation.* IEEE, 195–204.

[11] Mehdi Mirzaaghaei, Fabrizio Pastore, and Mauro Pezzè. 2012. Supporting test suite evolution through test case adaptation. In *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation.* IEEE, 231–240.

[12] Hoan Anh Nguyen, Tung Thanh Nguyen, Tien N Nguyen, and Hung Viet Nguyen. 2017. Interaction-based tracking of program entities for test case evolution. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME).* IEEE, 433–443.

[13] Terence Parr. [n. d.]. *ANTLR4.* https://www.antlr.org

[14] Leandro Sales Pinto, Saurabh Sinha, and Alessandro Orso. 2012. Understanding myths and realities of test-suite evolution. In *Proceedings of the ACM SIGSOFT 20th international symposium on the foundations of software engineering.* 1–11.

[15] José Miguel Rojas, Gordon Fraser, and Andrea Arcuri. 2015. Automated unit test generation during software development: A controlled experiment and think-aloud observations. In *Proceedings of the 2015 international symposium on software testing and analysis.* 338–349.

[16] Samiha Shimmi and Mona Rahimi. 2022. Leveraging code-test co-evolution patterns for automated test case recommendation. In *Proceedings of the 3rd ACM/IEEE International Conference on Automation of Software Test.* 65–76.

[17] Samiha Shimmi and Mona Rahimi. 2022. Patterns of Code-to-Test Co-evolution for Automated Test Suite Maintenance. In *2022 IEEE Conference on Software Testing, Verification and Validation (ICST).* IEEE, 116–127.

[18] Bart Van Rompaey and Serge Demeyer. 2009. Establishing traceability links between unit test cases and units under test. In *2009 13th European Conference on Software Maintenance and Reengineering.* IEEE, 209–218.

[19] Sinan Wang, Ming Wen, Yepang Liu, Ying Wang, and Rongxin Wu. 2021. Understanding and facilitating the co-evolution of production and test code. In *2021 IEEE international conference on software analysis, evolution and reengineering (SANER).* IEEE, 272–283.

[20] Robert White, Jens Krinke, and Raymond Tan. 2020. Establishing Multilevel Test-to-Code Traceability Links. In *IEEE/ACM International Conference on Software Engineering.*

[21] Andy Zaidman, Bart Van Rompaey, Serge Demeyer, and Arie Van Deursen. 2008. Mining software repositories to study co-evolution of production & test code. In *2008 1st international conference on software testing, verification, and validation.* IEEE, 220–229.

[22] Andy Zaidman, Bart Van Rompaey, Arie Van Deursen, and Serge Demeyer. 2011. Studying the co-evolution of production and test code in open source and industrial developer test processes through repository mining. *Empirical Software Engineering* 16, 3 (2011), 325–364.