2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER) | 978-1-7281-9630-5/20/$31.00 ©2021 IEEE | DOI: 10.1109/SANER50967.2021.00033

Understanding and Facilitating the Co-Evolution of Production and Test Code

Sinan Wang*, Ming Wen†, Yepang Liu*, Ying Wang‡, Rongxin Wu§
*Department of Computer Science and Engineering,
Guangdong Provincial Key Laboratory of Brain-inspired Intelligent Computation,
Southern University of Science and Technology, Shenzhen, China
†School of Cyber Science and Engineering, Huazhong University of Science and Technology, Wuhan, China
‡Software College, Northeastern University, Shenyang, China
§School of Informatics, Xiamen University, Xiamen, China
wangsn@mail.sustech.edu.cn; mwenaa@hust.edu.cn; liuyp1@sustech.edu.cn;
wangying@swc.neu.edu.cn; wurongxin@xmu.edu.cn

***Abstract*—Software products frequently evolve. When the production code undergoes major changes such as feature addition or removal, the corresponding test code typically should *co-evolve*. Otherwise, the *outdated test* may be ineffective in revealing faults or cause spurious test failures, which could confuse developers and waste QA resources. Despite its importance, maintaining such co-evolution can be time- and resource-consuming. Existing work has disclosed that, in practice, test code often fails to co-evolve with the production code. To facilitate the co-evolution of production and test code, this work explores how to automatically identify outdated tests. To gain insights into the problem, we conducted an empirical study on 975 open-source Java projects. By manually analyzing and comparing the positive cases, where the test code co-evolves with the production code, and the negative cases, where the co-evolution is not observed, we found that various factors (e.g., the different language constructs modified in the production code) can determine whether the test code should be updated. Guided by the empirical findings, we proposed a machine-learning based approach, SITAR, that holistically considers different factors to predict test changes. We evaluated SITAR on 20 popular Java projects. These results show that SITAR, under the within-project setting, can reach an average precision and recall of 81.4% and 76.1%, respectively, for identifying test code that requires update, which significantly outperforms rule-based baseline methods. SITAR can also achieve promising results under the cross-project setting and multiclass prediction, which predicts the exact change types of test code.**

***Index Terms*—Software evolution, test maintenance, mining software repositories**

I. Introduction

Testing is a widely-adopted technique for software quality assurance. In practice, software developers often write test code to validate whether the production code behaves as expected. Since software products frequently evolve, test code should co-evolve with the corresponding production code to remain effective in assessing software quality. Otherwise, the outdated test code may cause undesirable consequences.

Figure 1 shows an example of an outdated test from Apache `commons-pool` [1]. On June 13, 2011, a developer updated the production code to prevent the `_factory` attribute from being reset: when setting `_factory`, an `IllegalStateException`

```
  public void setFactory(Factory factory) throws Exception {
-   if (0 < getNumActive()) {
-     throw new IllegalStateException();
+   if (this._factory == null) {
+     this._factory = factory;
    } else {
-     clear();
+     throw new IllegalStateException();
    }
-   _factory = factory;
  }
```

(a) Production code change (commit `d2e27d1` by the developer *markt-asf* on June 13, 2011)

```
- @Test
- public void testSetFactory() {
-   GenericObjectPool pool = new GenericObjectPool();
-   pool.setFactory(new SimpleFactory());
-   pool.setFactory(new SimpleFactory());
- }
```

(b) Test code change (commit `ffe6735` by the developer *psteitz* on June 16, 2011)

Fig. 1: Outdated test example in `commons-pool` (simplified)

will be thrown if it is non-null. However, the corresponding test code, which resets the factory, was not updated until June 16, 2011. There are 17 commits between the changes of production and test code. During this period, the project developers who were unaware of the production code change might be confused by the `IllegalStateException` when running the outdated test to test the evolved production code.

From the example, we can see that it is important to keep the test code co-evolving with the production code. However, this may not be an easy task for developers in practice, especially when their project has a large code base maintained by geographically separate teams. Existing work [2] has pointed out that, test code often fails to co-evolve with production code due to three reasons: (1) lack of time and QA resources for test maintenance, (2) unawareness of the existence of tests for a particular functionality, and (3) lack of time to run all tests (thus preventing the discovery of outdated tests).

Since it is often non-trivial for developers to keep test code up to date, a technique that can automatically identify outdated tests and prompt developers to make updates could be helpful. This work aims to develop such a technique to facilitate the co-evolution of production and test code (*production-test co-*

* Yepang Liu is the corresponding author.

evolution for short) and we focus on the unit testing scenario.

Given a set of production code changes, it is challenging to automatically decide whether the corresponding unit tests should be updated or not. Although existing work has studied the associations between production and test code (e.g., [3]–[6]), there is little research that explicitly explores under what situations test code should co-evolve with production code. To fill this gap and gain a better understanding of the co-evolution practices, we conducted a large-scale empirical study on 975 open-source Java projects. We found that whether to update a unit test can be determined by various factors, such as the type of the language constructs modified in the production code and the complexity of the production code changes. It is not effective to adopt heuristics that consider only single factors (e.g., when the body of a method is changed, the corresponding unit test should be updated) to predict test changes. Based on this key observation, we propose a machine-learning based approach, SITAR, to help developers identify whether a unit test should be updated when they make changes to the corresponding production code. SITAR is able to efficiently learn a model that considers the interrelations between the changed code components from the historical commits of a project. The model can then help developers make test update decisions by holistically considering multiple factors, e.g., changes of method calling relations and return values in production code. We have implemented SITAR and evaluated it on 20 popular Java projects. The results show that SITAR is effective in predicting test changes. For example, it achieves an average precision and recall of 81.4% and 76.1% (the maximal precision and recall are 93.5% and 87.7%), respectively, under the within-project setting. Comparatively, the rule-based methods can only obtain an average precision of 51.8%, although the average recall, 90.5%, is higher than that of SITAR (this is understandable as the rule-based methods make predictions simply based on single factors).

To summarize, this work makes the following contributions:

- We performed a large-scale empirical study to understand the practice of production-test co-evolution in 975 open-source Java projects.
- We proposed a machine-learning based approach, SITAR, for automated identification of outdated unit tests to facilitate production-test co-evolution.
- We evaluated SITAR on 20 Java projects under different sample configurations and model settings, which demonstrated the effectiveness and usefulness of our approach.

To facilitate future research, we release the artifacts of our research at https://github.com/sqlab-sustech/Sitar-project.

II. PRELIMINARIES

A. Java Project Directory Layout and Naming Convention

Java projects managed by Maven mostly follow the *standard directory layout* [7], where the `src/main` directory stores production code files and the `src/test` directory stores test code files. When naming a unit test class, Java developers often follow a convention to name a test class by appending the word "Test" to the name of the corresponding production class as a suffix (or prefix). For example, below is a pair of production-test class files in commons-io [8]:

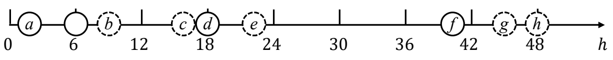
- `src/main/java/org/apache/commons/io/HexDump.java`
- `src/test/java/org/apache/commons/io/HexDumpTest.java`

Existing studies have widely leveraged the naming convention to associate production and test classes [9], [10]. It is shown that this simple approach can achieve a high precision and an acceptable recall in establishing production-test traceability [11]. In this work, we also use this approach to pair a production class with its corresponding unit test class.

B. Production-Test Co-Evolution

Co-evolution change pair: There are three types of file-level changes in most version control systems: *CREATE*, *EDIT*, and *DELETE*. A commit in a version control system typically contains one or more file changes and is assigned a unique hash ID (e.g., SHA1 in Git). A file f's change, denoted $change_f$, can be represented as a triple $(path_f,\ id_f,\ type_f)$, in which the three fields represent the relative path of f, the commit's hash ID, and f's change type, respectively. Based on this, we can define a *co-evolution change pair* as $(change_p,\ change_t)$, where $change_p$ denotes the change of a production code file p and $change_t$ denotes the change of the corresponding test code file t.

Simultaneous and postponed co-evolution: A unit test code file may be changed *simultaneously* with the corresponding production code file in a commit. The co-evolution may also be *postponed*, that is, the commit that changes the test code file is after the commit that changes the production code file. To ease understanding, let's consider an example.



On the timeline above, each circle represents a commit. For simplicity, let us assume that each commit only changes one file. Among these commits, suppose the solid circles $\{a, d, f\}$ change the production class `Foo`, the dashed circles $\{b, c, e, g, h\}$ change the corresponding test class `FooTest`, and the empty circle represents a commit that changes another class (neither `Foo` nor `FooTest`).[1] For subsequent discussions, we introduce two notations below.

First, given a commit co that modifies production code, we use $coevo_{ic}(co)$ to denote the set of ***co-evolved test changes*** that are postponed by at most i commits. Here, the subscript c means that we measure the postponement of co-evolution by the number of commits. With this notation, $coevo_{0c}(co)$ refers to the simultaneous test changes that occur in the commit co. For the above example, $coevo_{1c}(a)$ is an empty set, since the commit that immediately follows a (i.e., the empty circle) does not modify `FooTest`. For $i = 2, 3, 4, 5$, we have $coevo_{2c}(a) = \{b\}$, $coevo_{3c}(a) = \{b, c\}$, $coevo_{4c}(a) = \{b, c\}$, $coevo_{5c}(a) = \{b, c\}$, respectively. Note that $coevo_{5c}(a)$ does not contain e. This is because e is committed after d, which is a later change

[1]To ease presentation, we also use $a - h$ to represent the file changes.

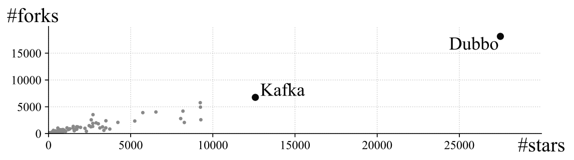


Fig. 2: Star and fork numbers of cloned projects

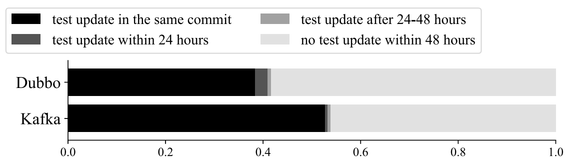


Fig. 3: The co-evolution situations in Dubbo and Kafka

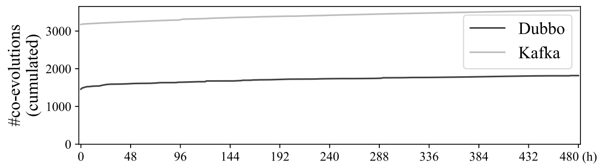


Fig. 4: Co-evolutions within 480 hours in Dubbo and Kafka

of the production class Foo. Hence e should be considered as a co-evolved test change of d, rather than that of a.

Second, we can also measure the postponement by the time duration. We use $coevo_{ih}(co)$ to denote the set of co-evolved test changes that are postponed by at most i hours after the commit co. For the above example, we have $coevo_{12h}(a) = \{b\}$ and $coevo_{24h}(a) = \{b, c\}$.

III. Understanding Production-Test Co-Evolution

In this section, we study the co-evolution change pairs in real-world Java projects. The study explores the following three research questions:

- **RQ1 (Prevalence of Co-Evolution):** *How prevalent is the phenomenon of production-test co-evolution?*
- **RQ2 (Change Type Combinations):** *What are the common change type combinations of production code and test code in co-evolution scenarios?*
- **RQ3 (Co-Evolution Characteristics):** *What kinds of production code changes often result in the co-evolution of the test code?*

For the study, we cloned all 975 Apache Software Foundation (ASF) Java projects from their GitHub mirrors on July 1, 2019. Figure 2 shows the star and fork numbers [12] of these projects, among which Dubbo [13] is the most popular project (27,498 stars and 18,142 forks) and Kafka [14] ranks the second (12,586 stars and 6,756 forks).

To answer RQs 1-2, we first discuss the results in the two most popular projects, i.e., Dubbo and Kafka, in detail and then present the results of all projects. To answer RQ3, we manually studied a subset of co-evolution change pairs and production code changes that did not result in co-evolution. We adopted the open-coding [15] process to categorize the studied changes to understand under what situations test code would co-evolve with the production code.

A. Prevalence of Co-Evolution

Figure 3 shows the co-evolution situations in two projects. For each project, the co-evolution changes are grouped by the time intervals between the commits of the production code and test code. The figure only reports the co-evolutions within 48 hours of the production code change for better visualization.

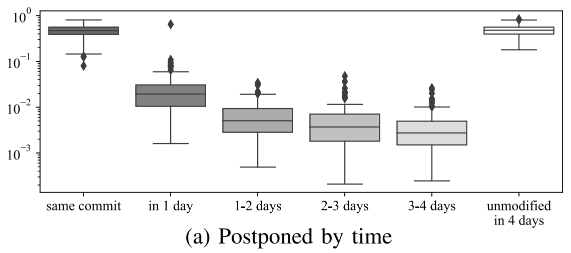
In Dubbo, nearly 40% test classes were changed in the same commit with the production class changes. In other word, 40% production class changes resulted in simultaneous co-evolutions. Most of the remaining production class changes did not result in test class co-evolution within 48 hours. Besides, postponed co-evolutions exist, but they only account for a small proportion. Also note that the postponed co-evolution within 24 hours (i.e., $coevo_{24h} \setminus coevo_{0h}$) is more frequent than that within 24-48 hours (i.e., $coevo_{48h} \setminus coevo_{24h}$). In Kafka, the proportion of simultaneous co-evolutions is even higher (over 50%), while the proportions of postponed co-evolutions is smaller. This shows that in well-maintained open-source projects like Dubbo and Kafka, developers tend to update the test code soon when there are important changes to the production code.

We also investigated the postponed co-evolutions that happened beyond 48 hours. The result is shown in Figure 4. Each point at time t represents the number of observed co-evolution change pairs in which the test code is updated within t hours of the production code changes. The point at zero refers to the number of simultaneous co-evolutions. In both plots of Dubbo and Kafka, the number of co-evolutions slightly increases after the first data point. This indicates that co-evolution rarely happen long after the production code is changed.

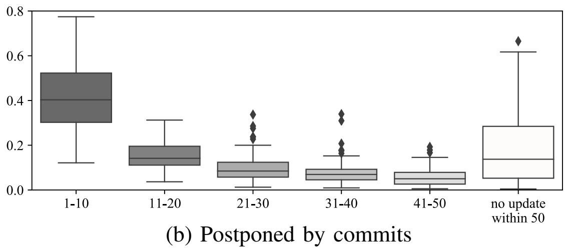
To understand the co-evolution practices in other ASF projects, we first collected their co-evolution change pairs. For each project, we categorized the pairs into six time intervals as shown in Figure 5(a) and represented the result as a six-dimensional vector.[2] We normalized each vector so that the values sum up to 1. Since many ASF projects are small in size and have few commits or unit tests, their corresponding vectors contain zero values. For better visualization, we remove those projects whose vector contains zero values. Figure 5(a) presents the distribution for each category among the studied ASF projects (after filtering, 151 projects remained). Similarly, we also categorized the co-evolution change pairs by the number of commits between the production code change and the test code change. The statistics are shown in Figure 5(b) (after filtering, 93 projects remained).

From Figure 5(a), we can observe that in most projects, co-evolutions often happen in the same commits, with a median proportion around 47.0% and a mean proportion of 46.9% (the vertical axis is in log scale). The number of postponed co-evolutions decreases by time in most projects, similar to

[2]Due to page limit, we only considered six time intervals in this paper. More detailed data are available on our project website.



(a) Postponed by time



(b) Postponed by commits

Fig. 5: Co-evolution statistics of the studied ASF projects

TABLE I: Co-evolution change types in Dubbo and Kafka

	Change Type								
	CC	**CD**	**CE**	**DC**	**DD**	**DE**	**EC**	**ED**	**EE**
Dubbo	403	0	2	0	247	2	100	14	941
Kafka	316	1	2	2	19	2	149	2	2,899

what we observed in Dubbo and Kafka. There is also a large amount (the median is 48.0% and the mean is 48.3%) of tests not being changed within four days. A similar conclusion can be drawn from Figure 5(b) (simultaneous co-evolutions are excluded as they are already presented in Figure 5(a)).

Finding 1: (1) Although production-test co-evolutions are common, a large proportion of production code changes would not result in the co-evolution of test code; (2) Postponed co-evolutions happen but simultaneous co-evolutions are much more frequent.

B. Change Type Combinations

Table I shows the distribution of co-evolution change types in Dubbo and Kafka. Here we only consider the co-evolutions that happen within 20 days of the production code changes. In the table, the change types **C**, **D**, and **E** represent "Creating", "Deleting", and "Editing", respectively. For each combination (e.g., **CC**), the first letter represents of the change type for the production code file and the second letter represents the change type for the test code file.

In both projects, the most frequent change type combination is **EE**, where developers update an existing production class and its test class. The second most frequent combination is **CC**, where developers add new production classes and create the corresponding test classes to test the new production code. Among the remaining combinations, **EC** is also frequent. In the **EC** scenario, developers update an existing production class and create a corresponding test class.

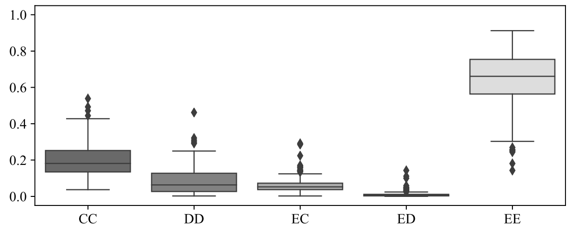


Fig. 6: The distribution of co-evolution change type combinations in 182 ASF projects

Besides the common combinations, we also found some counter-intuitive combinations. For example, there are two **DC** cases in Kafka. We checked the commits and found that both cases are related to the refactoring of production classes. Since these cases are rare, we do not make further discussions.

Figure 6 shows the distribution of co-evolution change type combinations in other studied ASF projects. Here we excluded the following four rarely observed combinations (see Table I): **CD**, **CE**, **DC** and **DE** and only kept 182 ASF projects that have all the five remaining combinations (data processing is similar to Figure 5). The figure shows that the proportion of **EE** is significantly larger than the other combinations, with a midian of 66.1%, and two quartiles of 56.4% and 75.4%, respectively. The overall distribution is close to that in Dubbo and Kafka (Table I).

Finding 2: (1) Various combinations of change types may occur in production-test co-evolution, some of which are counter-intuitive (e.g., **CD** or **DC**); (2) The change type combination **EE** is significantly more frequent than the other combinations.

C. Co-Evolution Characteristics

Knowing that co-evolutions often happen soon after production code changes, and there are different combinations of change types at the file level, we then went deep to investigate the co-evolved production and test code by inspecting the code changes. Based on the previous findings, if a test is changed within 48 hours of the production code changes, we regard the pair of production and test code as a "positive pair"; On the contrary, if the test is not changed within 480 hours of the production code changes, we regard the pair of production and test code as a "negative pair". From the 975 ASF projects, we extracted 210,291 positive pairs and 217,831 negative pairs. For each type of pairs, we randomly sampled a subset of production code changes that are **EDIT**s (we call such changes "*production patches*"), since the change type combination **EE** dominates the other combinations according to Finding 2. Specifically, for each type of pairs, we analyzed 384 randomly picked production patches, which can represent the population with a confidence level of 95% and a 5% margin of error.

Manual analysis of production patches: During the analysis, we tagged each patch by the location where it is applied. Note that a patch can involve code changes in multiple

TABLE II: The details of the changes made by production patches

Samples		Programming-Language Construct							Natural-Language Construct			Average Patch Size (Lines)	
		package-id	import-stmt	class-dec	method-sig	method-body	field	annotation	copyright	javadoc	comment	add	delete
positive	384	5/3	166/22	14/0	59/8	242/43	87/3	59/3	34/28	98/0	51/0	29.12 (19.83)	19.65 (13.55)
negative	384	0/0	132/18	9/4	32/8	250/97	57/7	44/9	5/3	87/0	33/4	11.37 (7.72)	6.98 (4.47)

[1] The notation "*a/b*" under category c means: there are a patches that apply on c, and b patches apply **solely** on c
[2] In the notation "a (b)", b is the average number of patch lines, excluding empty lines and natural-language constructs
[3] The shorthands "stmt" stands for "statement", "dec" for "declaration", and "sig" for "signature"

locations, such as field and method body. The tagging was performed by one author and validated by another author. The process involved several rounds. In each round, the tagging author presented the results in detail to the validating author. Then they discussed to revise the results, by splitting, removing, or merging some tags. For example, the tag "program header" was split into "package id" and "import statement". As shown in Table II, after rounds of discussion, the two authors reached consensus and classified the production patches into 12 types, which belong to two major categories: (1) changes to programming-language constructs and (2) changes to natural-language constructs. From the tagging results, we can make several interesting observations.

Observation 1: *In the sampled positive pairs (positive samples), the production patches tend to modify more different language constructs.*

Among the seven types of programming-language constructs, the changes to "package identifier" only appear in positive samples: indeed, when a Java class is re-packaged, its test class should also be updated; otherwise, the test classes are not compilable. For import statements, the positive samples involve more changes to them than negative samples.

There are a few production patches that modified the "class declaration" by updating the production class's super class (or interfaces) or changing the generic type parameters. As shown in the table, none of the positive samples changed only the class declaration and most of them (12/14) were coupled with method changes. Negative samples, however, may only modify the class declaration itself. This is understandable: if a class's super class (or interfaces) is changed and some methods need to be overridden, its corresponding test class should also be updated, to test the overridden methods. Conversely, if there is no methods to be overridden, the test often remains unchanged.

Methods in a production class encapsulate almost all business logics (in Java programming, field accesses are often wrapped inside getter and setter methods). In order to better understand how method changes affect co-evolution decision, we looked at the changes to "method signature" and "method body" separately. Unlike the traditional definition of "method signature", we also considered modifiers and return type as part of the signature. Method declarations in abstract classes or interfaces also belong to "method signature". As shown in the table, 59 patches modified method signatures in the production code, leading to co-evolutions. On the contrary, 32 method signature modifications did not result in co-evolutions. For the co-evolution cases, nearly one third of the production patches (17/59) either added or deleted method parameters, or changed parameter types (13/59) into incompatible classes, which would make test code uncompilable. For the cases where there was no co-evolution, 14 patches changed the parameters or return types into compatible classes, e.g., from `String` to `Object` or from `Collection` to `Set`. Other types of signature changes include changing modifiers, adding annotations, and etc., which are rare. As we expected, most production patches changed method bodies. However, we found slightly more negative samples modifying method bodies (250 against 242). More interestingly, we observed 97 negative samples that only modified method bodies. This indicates that predicting test changes by looking at whether method bodies are changed by production patches may not be an effective approach to identifying outdated tests.

For those patches that modified the member attributes (i.e., "field"), 87 patches led to co-evolutions, while 3.4% of them did not modify other language constructs. In the negative samples, this proportion becomes 12.3%, which also supports Observation 1 since more negative samples only modified a single language construct. We can draw a similar conclusion for the changes to "annotation" lines.

Observation 2: *Production patches in positive samples tend to modify more lines than those in negative samples.*

Table II shows the average size of the sampled patches. For positive samples, on average, the production patches added 29.12 lines and deleted 19.65 lines. As for negative samples, these numbers are 11.37 and 6.98, respectively. If we exclude empty lines, comments, Javadoc, and copyright notices, on average, the positive samples added 19.83 lines and deleted 13.55 lines, while the negative samples added 7.72 lines and deleted 4.47 lines. As we can see, all numbers in the positive samples are significantly greater than those in the negative samples. Under a one-tailed t-test, we can accept the hypothesis that "positive samples add/delete more lines than negative samples" (all p-values are less than 0.00001). This suggests that when a production patch modifies more lines of code, a co-evolution is more likely to happen.

Observation 3: *Changes to natural-language constructs can also lead to co-evolutions.*

Copyright notices generally appear at the beginning of a source file. In our studied positive samples, we found that 34 production patches modified copyright notices and 28 involved only copyright notice changes. The numbers are significantly higher than those in the negative samples. This seems to suggest that it may be useful to consider copyright notice changes as a factor in predicting test changes. However, in practice, changing copyright notices for a whole project can be easily realized in IDEs. It is not meaningful to build a tool to prompt developers to update copyright notices in test code.

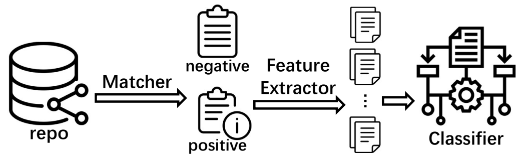


Fig. 7: SITAR's Workflow

As for Javadoc and comments, we found that their changes were often coupled with other types of changes. For example, in both positive and negative samples, no production patches changed only Javadoc. This indicates that considering the changes to such natural-language constructs may not be helpful in predicting test changes.

Finding 3: (1) Production code changes that result in the co-evolution of test code tend to modify different language constructs; (2) Production code changes involving more lines of code are more likely to result in the co-evolution of test code; (3) Changes to natural-language constructs are not good indicators of test changes.

Algorithm 1: Mining positive and negative samples

```
Input: {$commit_i$}: a code repository with $n$ commits
Output: pos_samples: positive samples (co-evolution),
        neg_samples: negative samples (not co-evolved)
pos_samples, neg_samples = ∅, ∅
for $i$ := 1 to $n$ do
    prod_list := production code changes in $commit_i$
    for $commit'$ in COMMITWITHIN(48, $commit_i$) do
        test_list := test code changes in $commit'$
        for prod in prod_list do
            test := CORRESPONDINGTEST(prod)
            if (test exists) && (test in test_list) then
                $p$ := (prod.path, $commit_i$.id, prod.type)
                $t$ := (test.path, $commit'$.id, test.type)
                add $(p, t)$ to pos_samples
    for $commit'$ in COMMITWITHIN(480, $commit_i$) do
        test_list := test code changes in $commit'$
        for prod in prod_list do
            test := CORRESPONDINGTEST(prod)
            if (test exists) && (test not in test_list) then
                $p$ := (prod.path, $commit_i$.id, prod.type)
                $t$ := test.path
                add $(p, t)$ to neg_samples
```

IV. Just-In-Time Test Update Prediction

A. An Overview of Our Approach

Our empirical study reveals that test code is often updated when certain types of production code changes occur. In real-world projects with a large number of production classes, keeping test code up to date whenever the production code changes can be a non-trivial task, especially when the production code and test code are maintained by different developers. To ease test maintenance in such scenarios, we propose a machine-learning based approach named SITAR for ju**S**t-**I**n-time **T**est upd**A**te p**R**ediction. SITAR automatically learns a model from the historical data of production-test co-evolution in a project and helps find outdated test code when the developers change the production code during the evolution of the project. For new projects without such historical data, the model can also be learned using data from other projects.

Figure 7 gives an overview of SITAR. Given a project's code repository, SITAR first extracts *features* from the production code changes and the corresponding *labels* from test code changes, i.e., their change types. Inspired by our empirical findings, when extracting features, SITAR considers not only various types of production code changes but also the size of such changes. Specifically, SITAR has three major components:

1) The ***Matcher*** analyzes the code repository and produces two types of samples: (1) positive samples, in which the test code co-evolves with the production code and (2) negative samples, in which the test code does not co-evolve with the production code.
2) The ***Feature Extractor*** converts samples produced by the Matcher into structural vector representations. For each sample, the feature is extracted from the "patch" and the original source code of the production classes. The change type of the corresponding test class is regarded as the sample's label.
3) The ***Classifier*** is trained using the historical data prepared by the previous steps. After training, the classifier can help predict whether a piece of test code should be updated when developers change the production code. To be practical, SITAR supports multiple learning algorithms and can be configured to perform multi-class classification, i.e., predicting the exact types of changes that should be made to the test code.

B. Mining Positive and Negative Samples

The Matcher mines samples in a code repository by matching production code changes and their corresponding test code changes in the repository's history. Following our empirical findings, if a test class is updated within 48 hours since its corresponding production class is changed, we regard it as a positive sample; On the other hand, if the test is not changed within 480 hours of the production code changes, we assume that no co-evolution will happen and regard it as a negative sample. Such a simple treatment may introduce noises into the mined samples. However, we did not observe any negative impact in our experiments.

Algorithm 1 shows how Matcher works. For each commit in the repository's main branch, Matcher finds all production file changes (line 3), which is trivial if the project follows the standard directory layout. To find positive samples, Matcher iterates over all commits within 48 hours after (and including) $commit_i$ (line 4), and collects test changes according to the standard directory layout (line 5). With the list of production code changes and the list of test code changes, Matcher then matches them according to the naming convention introduced in Section II (line 7). If the test exists and is in the change

TABLE III: Selected language constructs (features) and their corresponding grammars

Feature	Abbr.	Grammar Rule
package-id	P	*PackageDeclaration*
import-stmt	I	*ImportDeclaration*
class-dec[†]	C	*NormalClassDeclaration:* [^*ClassBody*]
param-list	L	*FormalParameterList*
cond-expr[‡]	D	*Statement:* `if` *(Expression) Statement* | `if` *(Expression) Statement else Statement* | `while` *(Expression) Statement* | `do` *Statement* `while` *(Expression);* | `for` *(ForInit; Expression; ForUpdate) Statement*
method-call	M	*MethodInvocation*
return-stmt	R	*ReturnStatement*
field	F	*FieldDeclaration*
annotation	A	*Annotation*

[†] All child nodes except *ClassBody* are considered
[‡] Only the child node *Expression* is considered

list (line 8), a co-evolution change pair is formed and added to the positive sample set (line 11). Mining negative samples is similar to mining positive samples and we do not further explain the process. Since the test code is not changed in a negative sample, it only records the test file path (line 18).

To determine test existence, Matcher stores all file names in the history and checks whether the test file ever appears. The rationale is that if the production code change does not immediately lead to the creation of a test, while the test is created long after that, it is reasonable to say that the production code change is not important enough to require a corresponding test, hence Matcher should regard this case as a negative sample.

Matcher labels samples by the file-level changes of the tests. There are two configurations. In the *binary class* configuration, samples are only labeled as **should change** (SC) or **should not change** (NC), depending on whether the sample is positive or negative. In the *multi-class* configuration, the positive samples are divided into three sub-categories: **should create** (SSC), **should edit** (SSE) or **should delete** (SSD).

C. Feature Extraction

The Feature Extractor converts each sample's (both positive and negative) production patch into structural vector representation. Particularly, it extracts the number of added/deleted lines of certain language constructs. Table III lists the selected features. Most concerned language constructs directly appear in Table II except that "method signature" is replaced by "parameter list", as most signature changes are actually related to changing parameters (Observation 1). Another difference is that "method body" is replaced by three representative constructs [3]: (1) conditional expression, (2) method call expression, and (3) return statement.

Previous work analyzes source code changes via AST differencing [3]. Unlike the existing work, which focuses on release-level changes, our work analyzes file-level changes at the granularity of commits. Thus our Feature Extractor can be implemented by a more light-weight method.

The Feature Extractor accomplishes its task via *interval stabbing* [16]. We say an interval $[l_i, r_i]$ is stabbed by a point j if $j \in [l_i, r_i]$. Given a set of interval I and a query point q, the interval stabbing problem is to find all the intervals in I that are stabbed by q. In a Java program, each language construct corresponds to an interval $[l, r]$, where l is the starting line number (left endpoint) and r is the ending line number (right endpoint). The intervals can be extracted from specific nodes in the program's parse tree (the grammar rules are given in Table III). For example, to obtain the interval of a while-loop condition, we can traverse the parse tree until the subtree "*Statement:* `while` *(Expression) Statement*" is reached and collect the line interval of the *Expression* tree node. Note that intervals may include each other [17] since language constructs may nest (e.g., conditional expressions may involve method calls). Given multiple changed lines (queried points), Feature Extractor finds out which language constructs (intervals) each line belongs to: for deleted lines, it queries the intervals generated by the original source code; for added lines, it queries the intervals generated by the patched code.

Given n queries, a brute force solution takes $O(n|I|)$ time, as all intervals are scanned for each query. This is inefficient since it checks for those unstabbed intervals repeatedly. These unnecessary operations can be eliminated by leveraging *segment tree* [18], whose query operations only take logarithmic time. The idea is to build an immutable segment tree from a parse tree and perform multiple queries. Since real-world projects are typically complex and large-scale, which can bring many intervals and query points, segment tree is an ideal optimization to the brute force solution.

D. Test Update Prediction

The Classifier can be implemented on top of various machine-learning algorithms. To be practical, it should also support both binary and multiclass classifications, since we may use both configurations depending on application scenarios. The Classifier can be trained using the commit history of a repository or other repositories. After training, given a new production patch, the Classifier will be able to predict whether its corresponding test should be updated (or should be created/edited/deleted, accordingly). In our implementation of SITAR, we used four classifiers that are widely-adopted for software maintenance tasks: Logistic Regression [19], Naïve Bayes [20], Random Forest [21], and Gradient Boosting [22]. In Section V-B, we will compare their performance.

V. EVALUATIONS

A. Experiment Setup

SITAR is mostly written in Python3 except that the Java parser is generated by the ANTLR metacompiler from the official g4 scripts [23]. For the Classifier component, we adopted the default implementations in scikit-learn [24] library.

To evaluate SITAR, we randomly selected ten ASF projects with a large number of positive/negative samples and seven

TABLE IV: Selected projects for evaluation

Project	From	#Co-evo	#Samples	#Commits	#Files	KLoC
ActiveMQ	ASF	563	1,452	10,197	4,455	415.0
CloudStack	ASF	1,274	3,237	32,008	5,404	613.9
Math	ASF	1,457	2,470	6,417	1,324	174.5
Flink	ASF	961	2,428	17,266	7,782	803.0
Geode	ASF	3,395	6,358	8,043	7,806	1,249.2
James	ASF	3,779	7,256	7,226	4,674	365.0
Log4j2	ASF	1,965	3,726	10,656	1,975	147.5
Storm	ASF	2,428	3,926	9,967	2,340	281.3
Usergrid	ASF	4,039	6,749	10,950	2,097	174.8
Zeppelin	ASF	1,395	2,527	4,124	758	124.6
JPacman	[10]	33	116	374	55	2.4
Gson	[3]	276	642	1,485	208	25.3
PMD	[3]	618	4,903	17,204	2,788	136.6
BioJava	[25]	2,401	2,998	6,208	1,322	150.9
IzPack	[25]	194	327	5,648	1,111	105.1
Joda-Time	[25]	338	565	2,152	330	86.5
dnsjava	[11]	172	247	1,911	258	29.4
Jackson	[26]	96	264	2,278	279	47.2
JRuby	[27]	489	1,220	48,720	1,716	259.3
jsoup	[28]	425	1,095	1,363	120	21.5

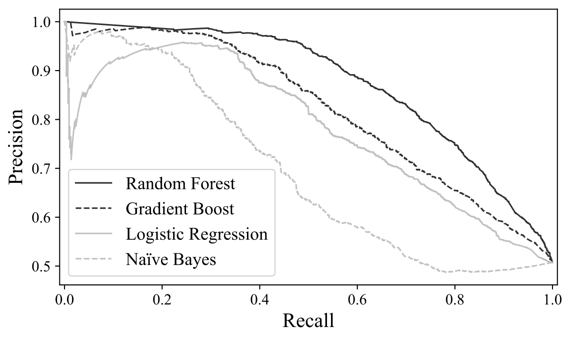


Fig. 8: Precision-recall curve of selected classifiers

projects widely used in co-evolution related research [3], [10], [11], [25]. We additionally selected three popular open-source projects, i.e., Jackson, JRuby, and jsoup, to enrich our subjects. The information of the 20 subjects is shown in Table IV. We will use them to conduct experiments to answer the following research questions:

- **RQ4**: *Which classifier is the most effective one?*
- **RQ5**: *Can* SITAR *effectively predict production-test co-evolutions in real-world projects?*
- **RQ6**: *Which features are most relevant to co-evolution?*
- **RQ7**: *Can* SITAR *outperform rule-based methods?*

B. RQ4: Comparison of Classifiers

To answer RQ4, we compared the aforementioned classifiers (Section IV-D) with the binary classification configuration. In this experiment, we used all data of the 20 projects (Table IV), and we balanced the sample label distribution by undersampling the majority of each project. To avoid over-fitting, we used the classifiers' default hyper-parameters set by the scikit-learn except that the Logistic Regression's `max_iter` is set to `1000` to converge iterative solving. We randomly split 90% of data for training classifiers and used the remaining 10% for validating the prediction results.

Figure 8 presents the precision-recall curves [29]. It shows that Random Forest classifier outperforms all other classifiers.

TABLE V: SITAR performance on selected projects (%)

	WP				WP			CP		
Project	Acc.	Prec.	Rec.	**Project**	Acc.	Prec.	Rec.	Acc.	Prec.	Rec.
ActiveMQ	71.8	75.6	64.8	JPacman	87.5	93.0	83.3	67.9	76.3	52.2
CloudStack	83.2	86.3	79.1	Gson	71.3	72.3	70.3	65.7	72.8	50.3
Math	71.1	72.8	67.7	PMD	74.0	74.2	73.8	62.9	67.9	49.0
Flink	84.4	84.5	84.3	BioJava	84.4	87.4	80.8	72.8	72.6	73.4
Geode	82.4	84.2	79.8	IzPack	71.8	72.2	72.8	71.4	73.8	66.2
James	75.2	76.7	72.6	Joda-Time	71.1	71.2	71.7	64.5	67.8	55.1
Log4j2	76.5	79.1	72.3	dnsjava	84.7	81.9	89.3	67.5	68.8	64.1
Storm	90.8	93.5	87.7	Jackson	68.3	67.2	75.0	59.7	65.3	41.5
Usergrid	86.1	89.0	82.5	JRuby	73.7	74.9	71.9	62.9	66.2	52.7
Zeppelin	71.6	72.5	70.0	jsoup	70.7	70.3	72.2	57.6	63.4	36.1
Avg.	79.3	81.4	76.1		75.7	76.5	76.1	65.3	69.5	54.1

"Acc.", "Prec.", "Rec." stand for accuracy, precision, and recall, respectively.

The Naïve Bayes classifier is less effective (its curve is at the bottom when the recall is over 0.18). The main reason for such a poor performance is that Naïve Bayes classifier assumes that all features are independent [20], which is not the case in our context: Finding 3 already shows that in a co-evolution scenario, the production patch tends to modify more language constructs, such as modifying the method signature and body simultaneously. The curve for the Logistic Regression classifier begins with a cliff. This indicates that it may estimate a high probability for a negative sample, resulting in false alarms of outdated tests. It is also worth mentioning that both ensemble classifiers (Random Forest and Gradient Boosting) are able to achieve a higher precision with the same recall, comparing to the other two classifiers.

Answer to RQ4: Random Forest outperforms other classifiers in test update prediction. Naïve Bayes and Logistic Regression classifiers are less effective.

C. RQ5: Performance of SITAR *in Real-world Projects*

To evaluate SITAR's performance in real-world projects, we conducted experiments with two settings, within-project (WP) and cross-projects (CP), using the Random Forest classifier and the binary classification configuration. We also performed undersampling for each project in binary classification. In the first setting, we performed 10-fold cross-validation on each project and calculated the average scores of each evaluation metric. In the second setting, we used all ASF projects for training and validated the performance of the resulting model with the newly-selected projects, and the final metric scores were averaged from 10 independent runs.

Table V lists the accuracy, precision, and recall scores in percentage. Among ASF projects under the WP setting, the accuracy ranges from 71.1% to 90.8% with an average score of 79.3%. The precision ranges from 72.5% to 93.5% with an average score of 81.4%. The average recall is 76.1%. The results indicate that SITAR can effectively identify outdated tests in ASF projects.

As for the new projects, SITAR obtains an average accuracy of 75.7%, precision of 76.5%, and recall of 76.1%. In some projects, such as BioJava and dnsjava, all scores are above 80%. This shows that SITAR can also perform well under the WP setting in non-ASF projects.

TABLE VI: Multiclass prediction results for the project Storm

	SSC	SSD	SSE	NC
SSC	969	19	26	28
SSD	19	176	7	3
SSE	12	1	986	185
NC	10	2	118	1,365

Under the CP setting, among the three metrics, precision is the highest in most projects. This indicates that SITAR can learn co-evolution characteristics from ASF projects, and the learned knowledge can be applied to the new projects for identifying outdated tests. However, the recall scores under the CP settings are less satisfactory. A possible reason is that the classifier is trained with solely the ASF data, thus it misses some practices that are not present in the ASF projects, resulting in too many false negatives in outdated test detection.

We also evaluated SITAR with multiclass prediction configuration (Section IV-B). We selected the top-3 ASF projects with the highest accuracies in previous experiments under the WP setting (Storm, Usergrid, and Flink) and run SITAR with their unbalanced multiclass data under the WP setting. The accuracies of the multiclass prediction are 89%, 85.4% and 86.2%, respectively. Table VI shows the confusion matrix of Storm. Each row is for the actual label while each column is for the predicted label. The confusion matrix shows that **SSC** and **SSD** can be accurately identified, as there are fewer false alarms to these two subtypes.

Answer to RQ5: SITAR exhibits promising performance, with an average accuracy, precision, and recall of 79.3%, 81.4%, and 76.1% in ten ASF projects. In the other ten projects, these average scores are all above 75%. SITAR can also perform multiclass prediction with a high accuracy.

D. RQ6: Contribution of Features

We use the recursive feature elimination (RFE) method [30] with the Random Forest classifier to rank the features. RFE recursively removes the least-important feature according to the classifier's performance until only one feature remains. The reversed removing order is then a ranking of the features' importance. With the feature ranking $f_1, f_2, \dots, f_n$ (from the most important to the least), we train multiple random forests in the following way to evaluate the contribution of the features. Specifically, for each feature f_i, we train random forests with two strategies: (1) the *prefix strategy* trains with features $f_1, f_2, \dots, f_i$ to evaluate the contribution of those important features and (2) the *postfix strategy* trains with $f_i, f_{i+1}, \dots, f_n$ to evaluate the contribution of those less significant features. To eliminate noise, we used the same train-validation partition for all random forests and initialized each model with an identical random seed. To ensure data significance, we selected eight projects in Table V whose WP accuracy is over 80%. We use 90% of the data for training and the remaining 10% for validating, and balance the label distribution for each project.

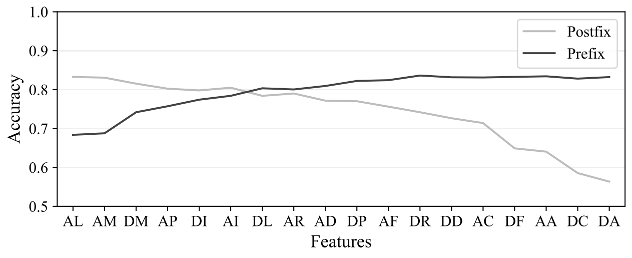


Fig. 9: The accuracy scores by different training features, features ranked according to their importance

Figure 9 shows the results. In the figure, the labels on the X-axis mean "Adding or Deleting certain language constructs". For example, AL means "adding parameter list lines" ('L' stands for parameter list as shown in Table III). The dark line is plotted according to the prefix strategy, thus its leftmost point is obtained by training with the feature AL alone. Similarly, the rightmost point of the grey line is obtained by training with only the feature DA. From the figure, we can see that using the feature AL alone can achieve an accuracy of nearly 70%, while using the feature DA alone can only achieve an accuracy of 56.4% (slightly better than random guesses). With the top-3 features (AL, AM and DM), SITAR can achieve an accuracy of 74.2%. In order to obtain such an accuracy with the postfix strategy, at least seven less-significant features, i.e., from DR to DA, are needed.

We also applied the RFE method on individual projects. The three features, AL, AM and DM, are all in the top-5 features for six out of the eight projects. For JPacman, only AL and AM are in the top-5 (note that JPacman has few samples). For dnsjava, AM and DM are in the top-5, while AL ranks at the 6th place. These results also suggest that AL (adding parameter list lines), AM and DM (modifying method call sites) are most relevant to co-evolution.

Answer to RQ6: Adding parameter list lines and modifying method call sites in the production code are most relevant to co-evolving the test code.

E. RQ7: Comparison with Rule-based Methods

We last compare SITAR with rule-based methods. According to Finding 1, the most trivial rule, which suggests test update whenever the production code is changed, would not work well in practice. Therefore, we chose not to compare with it. Our experiments compared SITAR with the baseline methods based on three co-evolution patterns mined by existing work [3]: (1) adding/deleting a production class would cause the corresponding test class to be added/deleted (R_1), (2) adding/modifying methods in production classes would result in adapting the test methods (R_2), and (3) modifying fields in production classes would lead to test update (R_3).

Table VII shows the performance of the rule-based methods when being applied on each project's balanced data. Due to page limit, the table only presents the data of the ASF

TABLE VII: Precision and recall of rule-based decisions (%)

	R_1		R_2		R_3		R_1+R_2+R_3	
	Prec.	Rec.	Prec.	Rec.	Prec.	Rec.	Prec.	Rec.
ActiveMQ	99.5	38.2	49.8	80.5	62.9	54.2	49.9	86.3
CloudStack	99.5	33.8	51.6	86.0	77.3	55.2	51.9	89.7
Math	97.6	24.6	53.7	78.8	55.4	39.5	52.0	86.8
Flink	90.1	14.2	51.6	96.7	70.3	64.2	51.6	97.9
Geode	99.6	30.2	55.6	89.7	72.2	42.9	54.5	92.2
James	100.0	24.8	52.0	87.1	74.2	48.4	51.2	89.1
Log4j2	98.3	19.9	55.9	88.4	65.1	44.6	54.3	91.5
Storm	99.7	38.9	52.6	90.7	73.8	70.4	51.5	92.5
Usergrid	96.7	42.2	49.7	76.1	68.2	53.0	50.5	82.9
Zeppelin	98.9	15.9	51.1	94.4	69.3	48.8	50.8	96.3
Avg.	98.0	28.3	52.4	86.8	68.9	52.1	51.8	90.5

projects. Among the rules, R_1 achieves the highest average precision but the lowest average recall. R_1 suggests test update whenever a production class is created or deleted. However, creating/deleting production classes is not as frequent as other kinds of code changes, and this is the primary reason of the low recall. Conversely, R_2 achieves a high recall but a low precision, as method changes cover most co-evolution and non-co-evolution scenarios (Table II). R_3 has a moderate performance comparing to R_1 and R_2 but both of its average scores are worse than that of SITAR under the WP setting. By combining the three baselines, we can achieve the highest average recall (90.5%) but its average precision is the worst.

Answer to RQ7: The performance of rule-based methods varies a lot, and SITAR significantly outperforms these methods.

VI. THREATS TO VALIDITY

The validity of our study may be subject to several threats. First, we only conducted our empirical study on ASF projects and our findings may not be generalizable to other software projects. To mitigate this threat, we used extra projects in the evaluation and found that our approach SITAR, which was designed based on the empirical findings, can also effectively detect outdated tests for non-ASF projects. Second, when investigating RQ3, we manually inspected 768 production patches. Such a manual process may be error-prone. To reduce the threat, we repeated our tagging process several times, and the results are cross-validated by two authors. Our dataset is also released for public scrutiny. Third, to obtain sample labels, we chose 48 hours for positive samples and 480 hours for negative samples (Section IV-B). These thresholds were set according to our empirical observations. In the future, we will study how to better identify samples, which may improve the performance of SITAR.

VII. RELATED WORK

A. Production-Test Co-Evolution and Traceability

Zaidman et al. studied the evolution patterns of production code and test code in two open-source projects [9] and one commercial project [31]. They presented their results with three visualization techniques. These visualizations, however, do not reflect the links between production code and test code. In comparison, our work presents how production and test code co-evolve in history. We also show the distribution of various co-evolution type combinations.

Levin and Yehudai studied co-evolution with semantic changes [4]. They were aware of the co-evolution caused by specific maintenance activities and used the number of *corrective*, *perfective*, and *adaptive* activities as code change features. Our work is complementary to their work as we focus on different co-evolution change types in practice, and we use syntactic features of code changes.

Various techniques [3], [10], [32]–[34] were developed to establish traceability links between production code and test code. Most of them are based on a set of pre-defined rules. Unlike them, we adopt the naming convention to build class-level production-test traceability links.

B. Machine Learning for Software Maintenance

Machine learning is widely used in various software maintenance tasks. An active research topic is software defect prediction (SDP) [35]. Recent SDP studies include automatic feature engineering [36], [37], cross-project learning [38], [39], addressing class imbalance problems [40], [41], classifier selection [42]. Besides SDP, there are many other application scenarios of machine learning techniques in the field of software maintenance. For example, in a recent survey [43], the authors studied 15 papers regarding code smell detection using machine learning and found that decision tree and support vector machine are the most commonly used algorithms. Other applications include, but are not limited to, regression test prioritization [44], code completion [45], and code clone [46].

VIII. CONCLUSION

In this work, we studied how production and test code co-evolve in real-world Java projects. We found that various factors can affect whether a test class should be updated, such as the types of modified language constructs and the complexity of the production code changes. Guided by our findings, we designed SITAR, a machine-learning based approach, to automatically identify outdated tests when the corresponding production code is changed. We evaluated SITAR on 20 Java projects and found that it can effectively identify outdated tests under the within-project setting. It can also achieve promising results under the cross-project setting and multiclass prediction. Moreover, SITAR can significantly outperform rule-based baseline methods proposed by existing work. In the future, we plan to study how to automatically extract more fine-grained syntactic and semantic features. This can help improve the effectiveness and practicability of our approach.

ACKNOWLEDGMENT

The authors would like to thank SANER 2021 reviewers for their comments and suggestions, which helped improve this paper. This work is supported by the National Natural Science Foundation of China (Grant Nos. 61932021, 62002125, 61802164, 61902056, and 61902329) and Guangdong Provincial Key Laboratory (Grant No. 2020B121201001).

References

[1] "apache/commons-pool: Mirror of apache commons pool." [Online]. Available: https://github.com/apache/commons-pool

[2] V. Hurdugaci and A. Zaidman, "Aiding software developers to maintain developer tests," in *2012 16th European Conference on Software Maintenance and Reengineering*. IEEE, 2012, pp. 11–20.

[3] C. Marsavina, D. Romano, and A. Zaidman, "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, 2014, pp. 195–204.

[4] S. Levin and A. Yehudai, "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, 2017, pp. 35–46.

[5] X. Sun, X. Peng, H. Leung, and B. Li, "Combort: A new approach for generating regression test cases for evolving programs," *International Journal of Software Engineering and Knowledge Engineering*, vol. 26, no. 06, pp. 1001–1026, 2016.

[6] D. Lo, L. Jiang, A. Budi *et al.*, "kbe-anonymity: test data anonymization for evolving programs," in *2012 Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*. IEEE, 2012, pp. 262–265.

[7] "Maven - introduction to the standard directory layout." [Online]. Available: http://maven.apache.org/guides/introduction/introduction-to-the-standard-directory-layout.html

[8] "apache/commons-io: Mirror of apache commons io." [Online]. Available: https://github.com/apache/commons-io

[9] A. Zaidman, B. Van Rompaey, S. Demeyer, and A. Van Deursen, "Mining software repositories to study co-evolution of production & test code," in *2008 1st international conference on software testing, verification, and validation*. IEEE, 2008, pp. 220–229.

[10] B. Van Rompaey and S. Demeyer, "Establishing traceability links between unit test cases and units under test," in *2009 13th European Conference on Software Maintenance and Reengineering*. IEEE, 2009, pp. 209–218.

[11] R. White, J. Krinke, and R. Tan, "Establishing multilevel test-to-code traceability links," in *42nd International Conference on Software Engineering (ICSE'20)*. ACM, 2020.

[12] N. Munaiah, S. Kroh, C. Cabrey, and M. Nagappan, "Curating github for engineered software projects," *Empirical Software Engineering*, vol. 22, no. 6, pp. 3219–3253, 2017.

[13] "apache/dubbo: Apache dubbo is a high-performance, java based open source rpc framework." [Online]. Available: https://github.com/apache/dubbo

[14] "apache/kafka: Mirror of apache kafka." [Online]. Available: https://github.com/apache/kafka

[15] K. J. Stol, P. Ralph, and B. Fitzgerald, "Grounded theory in software engineering research: a critical review and guidelines," in *Proceedings of the 38th International Conference on Software Engineering*, 2016, pp. 120–131.

[16] J. M. Schmidt, "Interval stabbing problems in small integer ranges," in *International Symposium on Algorithms and Computation*. Springer, 2009, pp. 163–172.

[17] A. Krokhin, P. Jeavons, and P. Jonsson, "Reasoning about temporal relations: The tractable subalgebras of allen's interval algebra," *Journal of the ACM (JACM)*, vol. 50, no. 5, pp. 591–640, 2003.

[18] J. L. Bentley, "Solutions to klee's rectangle problems," *Unpublished manuscript*, pp. 282–300, 1977.

[19] R. Wu, M. Wen, S. C. Cheung, and H. Zhang, "Changelocator: locate crash-inducing changes based on crash reports," *Empirical Software Engineering*, vol. 23, no. 5, pp. 2866–2900, 2018.

[20] R. T. Asmono, R. S. Wahono, and A. Syukur, "Absolute correlation weighted naïve bayes for software defect prediction," *Journal of Software Engineering*, vol. 1, no. 1, pp. 38–45, 2015.

[21] M. Kim, J. Nam, J. Yeon, S. Choi, and S. Kim, "Remi: defect prediction for efficient api testing," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, 2015, pp. 990–993.

[22] H. Zhu, L. Wei, M. Wen, Y. Liu, S. C. Cheung, Q. Sheng, and C. Zhou, "Mocksniffer: Characterizing and recommending mocking decisions for unit tests," 2020.

[23] "Java antlr grammar." [Online]. Available: https://github.com/antlr/grammars-v4/tree/master/java/java

[24] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[25] M. Alenezi, M. Akour, and H. Al Sghaier, "The impact of co-evolution of code production and test suites through software releases in open source software systems," *International Journal of Innovative Technology and Exploring Engineering (IJITEE)*, vol. 9, no. 1, pp. 2737–2739, 2019.

[26] "Fasterxml/jackson." [Online]. Available: https://github.com/FasterXML/jackson

[27] "jruby/jruby." [Online]. Available: https://github.com/jruby/jruby

[28] "jhy/jsoup." [Online]. Available: https://github.com/jhy/jsoup

[29] J. Guo, J. Cheng, and J. Cleland-Huang, "Semantically enhanced software traceability using deep learning techniques," in *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, 2017, pp. 3–14.

[30] I. Guyon, J. Weston, S. Barnhill, and V. Vapnik, "Gene selection for cancer classification using support vector machines," *Machine learning*, vol. 46, no. 1-3, pp. 389–422, 2002.

[31] A. Zaidman, B. Van Rompaey, A. van Deursen, and S. Demeyer, "Studying the co-evolution of production and test code in open source and industrial developer test processes through repository mining," *Empirical Software Engineering*, vol. 16, no. 3, pp. 325–364, 2011.

[32] Z. Lubsen, A. Zaidman, and M. Pinzger, "Using association rules to study the co-evolution of production & test code," in *2009 6th IEEE International Working Conference on Mining Software Repositories*. IEEE, 2009, pp. 151–154.

[33] L. Vidács and M. Pinzger, "Co-evolution analysis of production and test code by learning association rules of changes," in *2018 IEEE Workshop on Machine Learning Techniques for Software Quality Evaluation (MaLTeSQuE)*. IEEE, 2018, pp. 31–36.

[34] A. Qusef, G. Bavota, R. Oliveto, A. De Lucia, and D. Binkley, "Recovering test-to-code traceability using slicing and textual analysis," *Journal of Systems and Software*, vol. 88, pp. 147–168, 2014.

[35] I. H. Laradji, M. Alshayeb, and L. Ghouti, "Software defect prediction using ensemble learning on selected features," *Information and Software Technology*, vol. 58, pp. 388–402, 2015.

[36] S. Wang, T. Liu, and L. Tan, "Automatically learning semantic features for defect prediction," in *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*. IEEE, 2016, pp. 297–308.

[37] J. Li, P. He, J. Zhu, and M. R. Lyu, "Software defect prediction via convolutional neural network," in *2017 IEEE International Conference on Software Quality, Reliability and Security (QRS)*. IEEE, 2017, pp. 318–328.

[38] J. Nam, S. J. Pan, and S. Kim, "Transfer defect learning," in *2013 35th international conference on software engineering (ICSE)*. IEEE, 2013, pp. 382–391.

[39] G. Canfora, A. De Lucia, M. Di Penta, R. Oliveto, A. Panichella, and S. Panichella, "Multi-objective cross-project defect prediction," in *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*. IEEE, 2013, pp. 252–261.

[40] G. G. Cabral, L. L. Minku, E. Shihab, and S. Mujahid, "Class imbalance evolution and verification latency in just-in-time software defect prediction," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 666–676.

[41] M. Tan, L. Tan, S. Dara, and C. Mayeux, "Online defect prediction for imbalanced data," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 2. IEEE, 2015, pp. 99–108.

[42] D. Di Nucci, F. Palomba, R. Oliveto, and A. De Lucia, "Dynamic selection of classifiers in bug prediction: An adaptive method," *IEEE Transactions on Emerging Topics in Computational Intelligence*, vol. 1, no. 3, pp. 202–212, 2017.

[43] M. I. Azeem, F. Palomba, L. Shi, and Q. Wang, "Machine learning techniques for code smell detection: A systematic literature review and meta-analysis," *Information and Software Technology*, vol. 108, pp. 115–138, 2019.

[44] A. Bertolino, A. Guerriero, B. Miranda, R. Pietrantuono, and S. Russo, "Learning-to-rank vs ranking-to-learn: strategies for regression testing in continuous integration," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, 2020, pp. 1–12.

[45] S. Proksch, J. Lerch, and M. Mezini, “Intelligent code completion with bayesian networks,” *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 25, no. 1, pp. 1–31, 2015.

[46] G. Mostaeen, J. Svajlenko, B. Roy, C. K. Roy, and K. A. Schneider, “Clonecognition: machine learning based code clone validation tool,” in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019, pp. 1105–1109.