

https://dspace.org/

Matching Production and Test Files A Comparison of Filename and StatementBased Approaches

Gerald Kirui; Stephen Levitt

https://hdl.handle.net/10539/38885
SCOPUS

Downloaded from a DSpace repository. https://dspace.org/



Matching Production and Test Files: A Comparison of Filename and Statement-Based Approaches

Gerald Kipruto Kirui and Stephen Phillip Levitt(✉)

University of the Witwatersrand, Johannesburg, South Africa
stephen.levitt@wits.ac.za

Abstract. Accurate matching of test and production files plays a critical role in the analysis and evaluation of Test-Driven Development (TDD). However, current approaches often yield unsatisfactory results due to their reliance on filename-based matching. The purpose of this study is to compare the performance of a statement-based matching algorithm with the traditional filename-based approach. A comprehensive evaluation was conducted using 500 tests from 16 open-source Java projects, wherein the weighted F1-scores of both methods were assessed. Subsequently, the 95% confidence intervals were determined using a pseudosample size of 500. The statement-based approach achieved a 95% confidence interval of [0.6815, 0.7347], while the filename-based method had a notably lower interval of [0.1931, 0.2459]. These results demonstrate the superior performance of the statement-based matching algorithm, providing a more accurate and reliable solution for matching test and production files in TDD research. In conclusion, the statement-based matching algorithm significantly outperforms the filename-based method, which will benefit TDD research by offering a more accurate method of matching production files to test files.

Keywords: Traceability Links · Assertion Analysis · F1-Score

1 Introduction

Test-Driven Development or Test-Driven Design (TDD) is a software development practice in which automated tests are written prior to production code in an incremental, iterative manner with the sole purpose of delivering quality code [4].

Researchers who are interested in software testing are often interested in matching test code files with the corresponding production code files that are being tested, since TDD requires tests to be written at a certain point in time with respect to production code.

In the case of empirical TDD studies, this matching allows researchers to use file commit timestamps to determine when tests are written with respect to the production code being tested. Code coverage as an indicator of TDD

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2023
A. Gerber and M. Coetzee (Eds.): SAICSIT 2023, CCIS 1878, pp. 3–18, 2023.
https://doi.org/10.1007/978-3-031-39652-6_1

is insufficient, as it includes no timing information regarding when tests are written. Most researchers take it for granted that matching a test file to its target production files by comparing filenames is reasonably accurate. In this paper, we investigate the accuracy of such a method and propose an alternative, more accurate approach.

Section 2 discusses related work and outlines the problem statement. Sections 3 and 4 describe the methods of operation of the filename-based and statement-based matching algorithms, respectively. Sections 5 and 6 explain metrics to analyze the performance of the algorithms and strategies to quantify the confidence level in the performance measured, respectively. Section 7 provides and discusses the results. Lastly, Sect. 8 concludes the paper.

2 Background

2.1 Related Work

Zaidman et al. [15] investigated the co-evolution of production code and test code using three projects (two open-source projects and one industrial project). They analyzed commit behavior, the relative growth of production code and test code, and test coverage evolution over time. The authors concluded that TDD can be detected as the simultaneous committing of production code and associated test code in a VCS.

Borle et al. [1] compared the productivity and code quality of GitHub projects that use TDD to varying degrees. They created a TDD or TDD-like set (experimental) and a non-TDD set (control). The degree of TDD variation was allowed in two ways: the difference in commit timestamps between production code and test code, and class coverage. The authors analyzed the commit velocity of test-production file pairs and found that only 0.8% of Java projects strictly practice TDD and that there is no observable benefit from using TDD.

Kochhar et al. [5] conducted a study to determine the presence or absence of test cases in 50,000 GitHub projects. They found that 57.34% of projects did not have any tests, and 90% of projects had fewer than 100 tests. Projects with tests were larger in size than those without, with a weak positive correlation between the size of a project and the number of tests. There was also a negative correlation between the size of a project and the number of tests per LoC.

Hanhela [3] investigated the use of the test-first approach in open-source projects. The author found that only 5% of test-production file pairs were "test-first", while 61% were "test-with" and 34% were "test-last". The majority of projects (74%) had at least one test file, and 52% had at least one test-production file pair committed either before or with the production file. The author concluded that since the majority of test files are committed with their corresponding production files, they were likely written around the same time, and the test-first approach is not widely used in open-source projects.

Wang et al. [13] investigated the co-evolution of production code and test code. They analyzed 975 open-source Java projects to determine the factors that affect whether test code should be updated. From the results, they proposed a

machine-learning-based approach, SITAR, to predict the test code that requires an update when its corresponding production code is modified. After evaluating SITAR on 20 popular Java projects, they found that it obtained an average precision and recall of up to 81.4% and 76.1%, respectively, in a within-project setting.

Van Rompaey et al. [12] evaluated six strategies that resolve traceability between production code and test cases, namely naming convention, fixture element types, static call graph, last call before assert, lexical analysis, and co-evolution. To determine the baseline for traceability, they requested developers to manually select units under test that link to a given set of test cases from three projects, namely JPacman, ArgoUML, and Mondrian. The performance of each strategy was then determined via applicability and accuracy by comparing the output to the baseline. They concluded that the strategy based on naming convention yielded high precision and recall, although they note that it may be project-dependent. The authors recommended combining the strategies to improve traceability resolution.

White et al. [14] presented TCTracker, an approach to establish test-to-code traceability at the method-level and at the class-level. TCTracker combines multiple static and dynamic techniques to address the shortcomings of using a single technique for test-to-code traceability. They evaluated TCTracker on a dataset of traceability links that was manually curated from four open-source projects. They found that TCTracker performs better than any individual technique in establishing traceability links between tests and production code.

2.2 Problem Statement

Zaidman et al. [15] used the similarity of filenames to match a single test file to a single production file. Borle et al. [1] used a regular expression that matches imports from testing frameworks to check if a repository includes test files. They matched one test file to only one production file by checking if their filenames differed by the case-insensitive string "test". Kochhar et al. [5] identified test files by evaluating whether a file's name contains the string "test". Hanhela [3] matched one test file to only one production file according to the similarity in their filenames, i.e., the test file's name starts with the name of the production file and ends with the word "Test". Wang et al. [13] paired a production class with its corresponding test class by checking if the test filename differed from the production filename only by the string "Test" in the test filename. Van Rompaey et al. [12] established that linking a test case to its corresponding production file(s) by checking if they differ by the string "Test" can yield high precision and recall, depending on project guidelines. White et al. [14] noted that the techniques based on naming conventions yielded low recall.

Many test files are left unpaired when undertaking a purely name-based approach when matching test files and production files. For instance, suppose a project has three production files called `Circle.java`, `Rectangle.java`, and `Triangle.java` and one test file called `PerimeterTest.java`, which tests the implementation of the perimeter function in each of the production classes. If

the traditional filename-based matching algorithm were used to match the three production files to the test file, it would fail since the filenames are not defined similarly in order to explicitly link `PerimeterTest.java` to the production files. A statement-based matching algorithm would overcome this problem as it would not depend on the filenames but rather on the classes being instantiated in the test file. Moreover, the strategy used by most researchers to match tests to production files does not take into account integration tests because integration tests involve testing multiple software units as a combined entity [11] and it is not clear which unit the integration test should be named after.

This paper will introduce a strategy for matching a test file to its target files at statement-level granularity by linking assert statements in test code to the public methods of production classes that are tested by the assert statements.

3 Matching Files via Filenames

The filename-based algorithm checks if a filename contains the string "test" to determine if it is a test file. It checks if the production filename is at least 80% similar to the test filename after all filenames are converted to lowercase and "test" is removed from the test filename. Python's `difflib.SequenceMatcher` is used to match strings given a threshold and is based on the Ratcliff-Obershelp algorithm [7]. The similarity ratio is given by `2 * M/T` where `M` is the sum of the matches and `T` is the sum of the lengths of the original strings. Refer to Fig. 1: The production file `AnnotationValidatorFactory` would be matched to the test file `AnnotationsValidatorTest` because the similarity ratio is equal to 2 * (10 + 9)/(20 + 26) which is equal to 0.82.

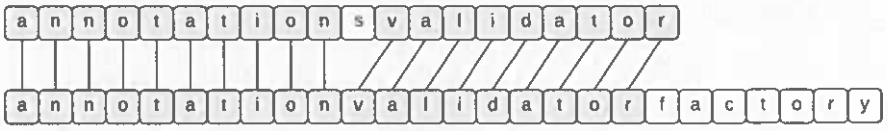


Fig. 1. Similarity between `annotationsalidator` and `annotationvalidatorfactory`

4 Matching Files via Assertions

The statement-based matching algorithm differs from the filename-based matching algorithm in that it analyzes source code line-by-line to match files, rather than matching files based on their filenames.

4.1 Distinguishing Between Tests and Production Files

PyDriller [8] is used to obtain a full list of production and test files in a project. To label a file as a test file, two factors are considered: whether the source code

includes at least one assert statement and whether there is at least one import from JUnit or TestNG, since they are the most popular testing frameworks [16]. The first regular expression in Listing 1.1 matches the assert statements used by the test frameworks. The second regular expression captures all the imports used in a file; the resultant list must be filtered by checking if the string "`junit`" or "`testng`" is part of the import, which would indicate that testing is performed in the file in question.

```
1 r"assert(Equals|That|ThatThrownBy|Same|NotSame|NotNull|Null
      |True|ArrayEquals|Throws|False|Timeout)\s*\("
2 r"import\s*.*;"
```

Listing 1.1. Regular expressions to determine whether a file is a test file

A file is considered a test file only if it has at least one match result for each of the regular expressions in Listing 1.1. Otherwise, it is considered a production file. The key subsystems utilized by the statement-based matching algorithm are depicted in Fig. 2 below.

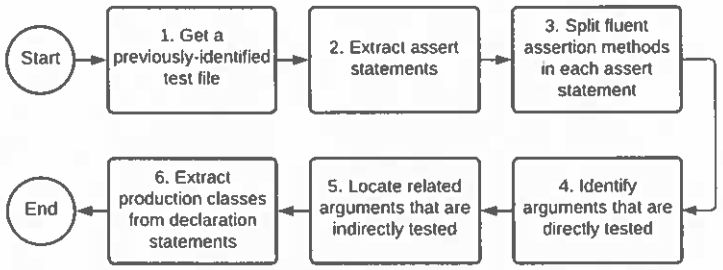


Fig. 2. Flow diagram of the statement-based matching algorithm.

4.2 Block 2: Extract Assert Statements

Block 2 takes test code as input and outputs a list of assert statements used in the test code. This block serves to isolate the parts of the test code that have a direct link to the production classes being tested. Assertion methods take objects of classes as arguments, and these could be direct instantiations of production classes or instantiations of third-party classes. A third-party class is a class that is provided by a third-party library or framework, which is a piece of software that is created and maintained by a separate organization or individual. Consider the test function in Listing 1.2, which verifies the behaviour of the add function from the Complex class from Apache Commons Math [10]. Given this test function, block 2 will return the array in Listing 1.3.

```
@Test
public void testAddComplexNumbers() {
    Complex a = new Complex(3, 4);
    Complex b = new Complex(1, -2);
    Complex expectedResult = new Complex(4, 2);
    Complex result = a.add(b);

    assertEquals(expectedResult, result);
}
```

Listing 1.2. Testing the Complex class from Apache Commons-Math

```
[
   "assertEquals(expectedResult, result);"
]
```

Listing 1.3. Output of block 1 given Listing 1.2

4.3 Block 3: Split Fluent Assertion Methods in Assert Statements

A fluent assertion method is a method that allows one to chain multiple assertions together such that the entire assert statement is more human-readable and expressive [6]. The functions `isNotEmpty`, `hasSize` and `contains` in the assert statement in Listing 1.4 are examples of fluent assertion methods. Block 3 is responsible for splitting the assertion methods with the goal of isolating the arguments in each assertion method. It takes a list of assert statements obtained from block 2 as input and returns the split fluent assertion methods. The element in the array in Listing 1.4 is an example of an assert statement that uses fluent assertion methods. When it is passed to block 3, the output is given by Listing 1.5.

```
[
    "assertThat(numbers).isNotEmpty().hasSize(5).contains(4)
    ;"
]
```

Listing 1.4. An example of an assert statement that uses fluent assertion methods

```
[
    "assertThat(numbers)",
    "isNotEmpty()",
    "hasSize(5)",
    "contains(4)"
]
```

Listing 1.5. The output of block 3 given Listing 1.4 as input

Notice that the block isolates each fluent assertion method from the assert statement by splitting them into separate elements.

4.4 Block 4: Identify Directly-Tested Arguments

Block 4 identifies the arguments used in the split assertion methods. It takes the split assertion methods of an assert statement as input and returns the arguments within these methods because some of the arguments may be objects instantiated by production classes, thus providing a clear link between the test file and a production file. The array of elements in Listing 1.6 is an example of an input to block 4. The corresponding output is given by Listing 1.7. Notice that only the arguments that are valid object names (`firstList` and `secondList`) are returned, whereas primitive data types and methods invoked by objects (such as 9 and `getSize` respectively) are discarded.

```
1 [
2     "assertThat(firstList)",
3     "isNotEmpty()",
4     "hasSize(secondList.getSize())",
5     "contains(9)"
6 ]
```

Listing 1.6. An example of split assertion methods and their arguments

```
1 [
2     "firstList",
3     "secondList"
4 ]
```

Listing 1.7. Output of block 4 given Listing 1.6 as input

4.5 Block 5: Locate Directly and Indirectly Tested Objects

Block 5 is responsible for identifying *and* locating the positions and names of object references within the code that have dependencies on the object references that are directly tested in assert statements. By taking into account the names of objects passed directly into assert statements (the output of block 4), it conducts a comprehensive analysis of the test code to determine all relevant relationships between objects. To achieve this, block 5 pinpoints the connections between different objects in the code, as one object could be the return value of another object.

Block 5 takes two parameters as input and returns one entity. The two parameters are the test file parsed as text and all the object references that are directly tested. The output is a set of the very same directly tested object references as well as all the object references they depend on. This is achieved by splicing the declaration statements associated with each object reference and then extracting the object from which it was instantiated, if it exists. Note that for each newly discovered object, there may exist yet another object from which it is instantiated. Hence, this subsystem is executed recursively until all the object references (both directly and indirectly tested) are identified. Consider Listing

1.8 for an example of the action of block 5. Note that Line 7 in Listing 1.8 represents the continuation of the test in the same fashion, i.e., the creation of `objectC`, `objectD`, etc.

```
1 @Test
2 public void someTestFunction() {
3     ProductionClassA objectA = new ProductionClassA();
4
5     ProductionClassB objectB = objectA.someFunctionA(1,
      someParameterA);
6
7     // Test continues ...
8
9     ProductionClassZ objectZ = objectY.someFunctionY(
      someParameterX);
10
11    assertNotNull(objectZ);
12 }
```

Listing 1.8. A test function where the object being tested is dependent on other objects

```
1 [
2     "objectA", "objectB", ..., "objectY", "objectZ"
3 ]
```

Listing 1.9. Output of block 5 given Listing 1.8 and "objectZ" as input

Notice that although `objectZ` is the only object that is directly tested, all other objects that it depends on for its creation are also returned by block 5. This is because the test would have failed if there were any issues during the creation of the other objects. Hence, the other objects are considered to be tested, albeit indirectly.

4.6 Block 6: Extract Production Classes from Declaration Statements

Block 6 is responsible for extracting the production classes from the declaration statements. Before the extraction, it identifies the lines of code that contain the declaration statements for all the names of objects that are directly or indirectly tested in an assert statement. Block 6 takes as input a collection of tested objects (the output of block 5) and a complete set of production classes in a project. It returns a subset of the production classes that appear in at least one declaration statement.

To achieve this, block 6 gets the positions of the directly and indirectly tested objects in the parsed test file from block 5. Then, for each position, a part of the parsed test code is extracted via string slicing. The extracted substring is the declaration statement corresponding to the object whose position was provided. The start of each substring is the last newline character ("\n") before the given

position, and the end of each substring is the first semi-colon (";") after the given position. Lastly, each substring is analyzed to check for the presence of production class names.

Consider Listing 1.10: the `testPolynomialFunction` method examines the behavior of two distinct polynomial function classes, `PolynomialFunction` and `PolynomialFunctionLagrangeForm` from Apache Commons Math. Initially, a `PolynomialFunction` object named `func` is instantiated using an array of coefficients $\{1, -2, 1\}$, representing the polynomial $f(x) = (x-1)^2$. The method computes the value of this polynomial at $x = 2.0$, storing the result in the variable `resultOne`. Next, a `PolynomialFunctionLagrangeForm` object called `lagrange` is created, utilizing arrays of x-coordinates $\{-1, 0, 1\}$ and corresponding y-coordinates $\{2, 1, 0\}$. The method then calculates the value of the Lagrange polynomial at $x = 2.0$ and assigns it to the variable `resultTwo`. Finally, the test function employs assert statements to validate the correctness of both polynomial evaluations. Listings 1.11, 1.12 and 1.13 show the output of each block from 2 to 6 upon analyzing this test code. The positions of object names identified by block 5 are indicated in yellow in Listing 1.10.

```
1 public void testPolynomialFunction() {
2     double[] coeffs = {1, -2, 1};
3     PolynomialFunction func = new PolynomialFunction(
      coeffs);
4     double resultOne = func.value(2.0);
5
6     double[] x = {-1, 0, 1};
7     double[] y = {2, 1, 0};
8     PolynomialFunctionLagrangeForm lagrange = new
      PolynomialFunctionLagrangeForm(x, y);
9     double resultTwo = lagrange.value(2.0);
10
11    assertThat(resultOne).isEqualTo(1.0)
12                    .and(resultTwo).isEqualTo(-1.0);
13 }
```

Listing 1.10. Testing the behaviour of classes from Apache Commons-Math

```
1 ["assertThat(resultOne).isEqualTo(1.0).and(resultTwo).
    isEqualTo(-1.0);"] // block 2
2 ["assertThat(resultOne)", "isEqualTo(1.0)", "and(resultTwo)
    ", "isEqualTo(-1.0)"] // block 3
3 ["resultOne", "resultTwo"] // block 4
4 ["resultOne", "resultTwo", "func", "lagrange"] // block 5 -
     all object names associated with the test
5 [96, 268, 332, 433, 146, 344, 381, 158] // block 5 -
    corresponding positions of the object names
```

Listing 1.11. The output of block 2 - 5

```
[
    "double resultOne = func.value(2.0);",
    "PolynomialFunction func = new
    PolynomialFunction(coeffs);",
    "double resultTwo = lagrange.value(2.0);",
    "PolynomialFunctionLagrangeForm lagrange = new
    PolynomialFunctionLagrangeForm(x, y);"
]
```

Listing 1.12. Declaration statements extracted by block 6

```
[
    "PolynomialFunction",
    "PolynomialFunctionLagrangeForm"
]
```

Listing 1.13. Production classes identified by block 6

5 Performance Metrics: Precision, Recall and F1-Score

A true positive (TP) production file selection is when a production class is accurately identified by the algorithm in a test file, while a false positive (FP) production file selection is when the algorithm falsely reports that a production class is tested. A false negative (FN) production file selection is when a production class is tested by a test file, but the algorithm fails to identify it. The performance of the file matching algorithms on a single project is determined using precision (ρ), recall (λ) and F1-score (ψ) according to the expressions below:

$$\rho = \frac{TP}{TP + FP} \tag{1}$$

$$\lambda = \frac{TP}{TP + FN} \tag{2}$$

$$\psi = 2 \cdot \frac{\rho \cdot \lambda}{\rho + \lambda} \tag{3}$$

The ground truth (a set of production files that are truly tested) for a single test file was determined by manually analyzing each file and recording the tested production classes. The manual analysis was done by a single individual; a script that randomly samples one test file per execution was used. After every execution, the script also provided a subset of production filenames that appear in the test file (that may or may not be tested). The individual then manually looked through the code to determine which production files from the subset were actually tested by the test file. The TP, FP, and FN values were evaluated by running a script that compared the output of the algorithms to the ground truth. For a more general indication of the performance of the file

matching algorithms, multiple projects were analyzed. Hence, the weighted precision, weighted recall, and weighted F1-score were used to represent the overall performance, where the weight is determined by the number of test files in each project. If there are n projects with ω_i tests in the i^{th} project, then the weighted precision ($\bar{\rho}$), weighted recall ($\bar{\lambda}$), and weighted F1-score ($\bar{\psi}$) are given by the following expressions:

$$\bar{\rho} = \frac{\sum_{i=1}^{n} \omega_i \cdot \rho_i}{\sum_{i=1}^{n} \omega_i} \tag{4}$$

$$\bar{\lambda} = \frac{\sum_{i=1}^{n} \omega_i \cdot \lambda_i}{\sum_{i=1}^{n} \omega_i} \tag{5}$$

$$\bar{\psi} = \frac{\sum_{i=1}^{n} \omega_i \cdot \psi_i}{\sum_{i=1}^{n} \omega_i} \tag{6}$$

6 Bootstrap Confidence Intervals

The weighted F1-score was chosen to represent the overall performance of the matching algorithms. The greater the number of test files used to evaluate the performance of the matching algorithms, the greater the confidence level in the weighted F1-score, but it is impossible to collect an infinite number of test files. Hence, the bootstrap confidence interval for the weighted F1-score was chosen to quantify the level of certainty in the performance of the matching algorithms when a given sample size is used. It is a method for estimating the certainty in the weighted F1-score due to sampling variation by resampling the dataset with replacement and computing the weighted F1-score for each bootstrap sample. The confidence interval provides a range of likely values for the true population value of the weighted F1-score. There are three main steps for determining the confidence interval of a metric [2]:

1. Create a distribution based on K bootstrapped samples for the statistic u.
2. Sort the values in ascending order.
3. The lower bound of the 100 $(1-2\alpha)$ percent confidence interval is the K α^{th} value, and the upper bound is the K $(1-\alpha)^{th}$ value in the sorted distribution.

K represents the number of pseudosamples that are generated from the original sample, u represents the statistic of interest (e.g., mean), and α represents a variable that determines the confidence level (e.g., if α equals 0.025, then the confidence level is 95%). Bootstrap confidence intervals are valuable in this study as they inform the reader of the confidence in the reported performance of the algorithm.

7 Results and Discussion

The performance of the statement-based and filename-based matching algorithms was evaluated using 500 tests from sixteen open-source Java projects. Figure 3 shows the F1-scores of the two matching algorithms for each Java project. Table 1 shows the weighted performance metrics of the matching algorithms. Table 2 shows a summary of the statistics related to the F1-score of the Java projects.

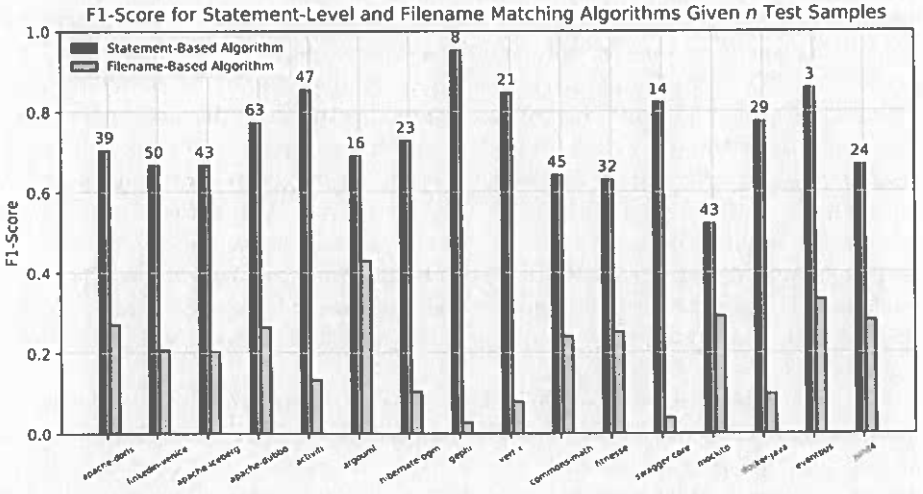


Fig. 3. F1-scores for the statement-based and filename-based matching algorithms given n test samples from different Java projects.

Table 1. Weighted performance metrics for the matching algorithms.

Matching Algorithm	Weighted Precision (ρ)	Weighted Recall ($\bar{\lambda}$)	Weighted F1-Score ($\bar{\psi}$)	Weighted Accuracy
Statement-Based	0.8667	0.6114	0.7170	0.7427
Filename-Based	0.4947	0.1401	0.2180	0.3803
Performance Improvement	+0.3720	+0.4713	+0.4990	+0.3624

The statement-based matching algorithm's weighted precision (86.67%) is 37.2% points higher than the filename-based matching algorithm's weighted precision (49.47%). Practically, this means that the statement-based matching algorithm has a 37.2% higher likelihood of correctly identifying tested production files while minimizing the inclusion of untested production files. Similarly, the statement-based matching algorithm's weighted recall (61.14%) exceeds

Table 2. Summary of statistics across the 16 Java Projects

Matching Algorithm	Min	Max	Variance
Statement-Based	0.5206	0.9517	0.0121
Filename-Based	0.0256	0.4286	0.0129

the filename-based matching algorithm's weighted recall (14.01%) by 47.13% points. This indicates that the statement-based matching algorithm has a 47.13% higher likelihood of capturing tested production files, significantly reducing the chances of missing such files compared to the filename-based matching algorithm. Regarding the F1-score, the statement-based matching algorithm's weighted F1-score (71.70%) is 49.9% points higher than the filename-based matching algorithm's weighted F1-score (21.80%). This suggests that the statement-based matching algorithm has a 49.9% better balance between precision and recall, making it more effective at accurately identifying tested production files while minimizing both false positives and false negatives. In practical terms, these percentage differences imply that the statement-based matching algorithm is far more effective at matching test files to production files, resulting in more accurate and reliable outcomes if it is used for TDD research. However, it incurs higher spatial and time complexity, requiring more computer memory and operations, leading to a longer processing time.

Consider the performance of the algorithms on the project Gephi: the extreme performance difference between the algorithms may be due to the sampled tests being integration tests, as each test tests an average of ten production files; hence, the chosen test filename does not match any one particular production file. Since the statement-based matching algorithms analyzes source code instead of filenames, its performance is not affected in the way that the filename-based matching algorithm is affected. Notice the relatively poor performance of the statement-based matching algorithm with respect to Mockito. Mockito, as a mocking framework, is designed to simplify the process of creating mocks and stubs for unit testing [9], making it highly likely that its own tests rely on the features it provides, such as dependency injection, mocking, and stubbing. Mockito's own tests primarily consist of unit tests that focus on testing individual components in isolation, using mocks and stubs to replace external dependencies. Although some integration tests might be employed to ensure proper functioning of components when interacting with each other, the primary focus is on unit testing with mocks and stubs, allowing the Mockito development team to ensure the framework's reliability and correctness while maintaining a fast and efficient testing process. Since the statement-based matching algorithm's method of operation does not detect the use of mocks or stubs, this may explain its relatively poor performance with respect to Mockito.

The difference between the upper bound and lower bound values (expressed as a percentage) for the 95% confidence interval of the weighted F1-score for both matching algorithms is displayed in Fig. 4. Table 3 shows upper bound and lower bound values using a sample size of 10 and 500.

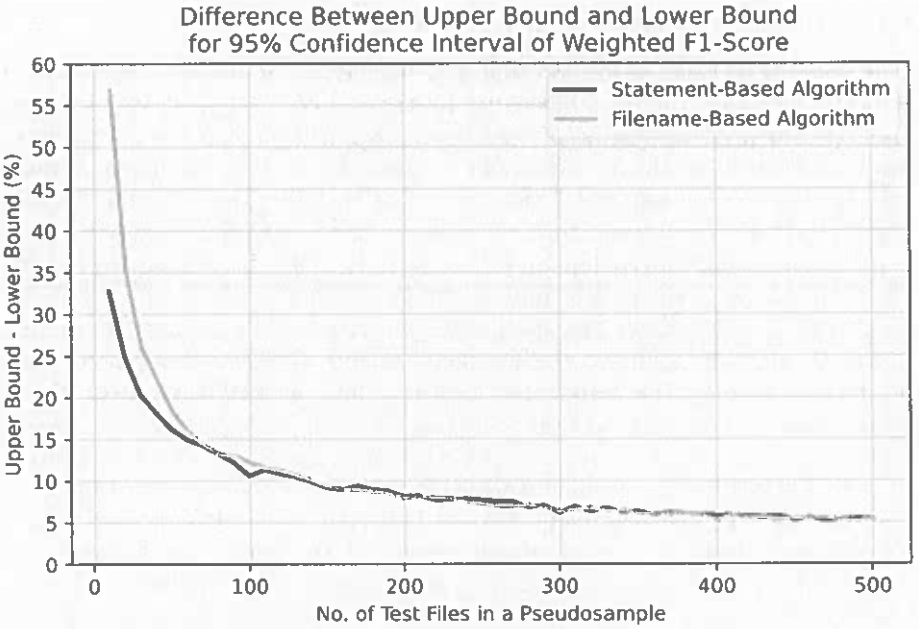


Fig. 4. Difference between the upper bound and lower bound for the 95% confidence interval of the weighted F1-score for the matching algorithms

Table 3. Upper bound and lower bound values for the weighted F1-score for the matching algorithms for 10 test samples and 500 test samples

Matching Algorithm	$\bar{\psi}_{upper}$ $(n = 10)$	$\bar{\psi}_{lower}$ $(n = 10)$	$\bar{\psi}_{upper}$ $(n = 500)$	$\bar{\psi}_{lower}$ $(n = 500)$
Statement-Based	0.8952	0.5676	0.7347	0.6815
Filename-Based	0.7501	0.1816	0.2459	0.1931

Pseudosample sizes ranging from 10 to 500 in increments of 10 were used to generate the plot. At a pseudosample size of 500, the 95% confidence interval of the statement-based matching algorithm is [0.6815, 0.7347] and that of the filename-based matching algorithm is [0.1931, 0.2459]. This suggests that, regardless of the specific sample size used, the statement-based matching algorithm is more likely to provide more accurate matches between test files and production files due to its higher weighted F1-score confidence interval. Moreover, the differences in the lower bounds (0.4884) and the upper bounds (0.4888) indicate a substantial performance gap in favor of the statement-based matching algorithm. This consistent superiority of the statement-based matching algorithm's confidence intervals suggests that it is more favourable to use for matching test files to production files in most scenarios.

7.1 Threats to Validity

This study may have limitations due to a sample size of 500 tests from sixteen projects, although this was mitigated by the use of the bootstrap confidence interval. The implementation of the filename-based algorithm from prior literature could not be obtained; there could be variations in the performance between the algorithm implemented for this study and the algorithm in prior research. The manual analysis of test files to determine which production classes are tested may have also introduced human error. To reduce the manual labour and the chance of human error in determining the production classes that are tested, a script was used to simply list the production classes that appear anywhere in the test code. This reduced the likelihood of incorrect classification of tested production classes. The statement-based matching algorithm may be affected when applied to new datasets that rely heavily on mocks or stubs, as it was not designed to consider the presence of mocks or stubs. The external validity of the findings may be constrained because of the use of open-source Java projects since software projects written in different languages or in different contexts may have different testing or programming styles. TDD involves writing tests prior to production code, and the statement-based matching algorithm can be used to detect the use of TDD. However, other aspects, such as the time difference between committing tests and production files, need to be taken into account to accurately detect the use of TDD. Hence, the statement-based matching algorithm alone may not accurately assess whether TDD is being practiced since it analyzes only one aspect of TDD.

8 Conclusion

This paper has proposed a statement-based matching algorithm. It consists of six blocks: block 1 gets a previously-identified test file; block 2 extracts assert statements; block 3 splits fluent assertions; block 4 identifies directly tested arguments; block 5 identifies directly and indirectly tested objects; and block 6 extracts production classes from declaration statements. The statement-based matching algorithm was shown to be more accurate, as it has a weighted F1-score of 0.7170 and a 95% confidence interval of [0.6815, 0.7347], whereas the filename-based matching algorithm has a weighted F1-score of 0.2183 and a 95% confidence interval of [0.1931, 0.2459]. The improved accuracy will benefit future TDD research as it can be used as a more accurate way to check if a project has used TDD, increasing the validity of the conclusions drawn by researchers. Future work will extend the statement-based matching algorithm to detect static classes in addition to instance classes, thereby improving recall.

References

1. Borle, N., Feghhi, M., Stroulia, E., Grenier, R., Hindle, A.: Analyzing the effects of test driven development in GitHub. In: 2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE), pp. 1062–1062. IEEE (2018)
2. Cohen, P.R.: Empirical Methods for Artificial Intelligence, vol. 139. MIT Press, Cambridge (1995)
3. Hanhela, J.: Usage of test-driven development in open source projects. Ph.D. thesis, Master's thesis Jari Hanhela Spring (2015)
4. Janzen, D., Saiedian, H.: Test-driven development concepts, taxonomy, and future direction. Computer **38**(9), 43–50 (2005)
5. Kochhar, P.S., Bissyandé, T.F., Lo, D., Jiang, L.: Adoption of software testing in open source projects-a preliminary study on 50,000 projects. In: 2013 17th European Conference on Software Maintenance and Reengineering, pp. 353–356 (2013). https://doi.org/10.1109/CSMR.2013.48
6. Leotta, M., Cerioli, M., Olianas, D., Ricca, F.: Fluent vs basic assertions in java: an empirical study. In: 2018 11th International Conference on the Quality of Information and Communications Technology (QUATIC), pp. 184–192. IEEE (2018)
7. Ratcliff, J.W., Metzener, D.E.: Pattern-matching-the gestalt approach. Dr Dobbs J. **13**(7), 46 (1988)
8. Spadini, D., Aniche, M., Bacchelli, A.: Pydriller: python framework for mining software repositories. In: Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, pp. 908–911 (2018)
9. Spadini, D., Aniche, M., Bruntink, M., Bacchelli, A.: To mock or not to mock? an empirical study on mocking practices. In: 2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR), pp. 402–412. IEEE (2017)
10. The Apache Software Foundation: Apache Commons Math GitHub Repository. https://github.com/apache/commons-math. Accessed 2022
11. Thomas, D., Hunt, A.: Mock objects. IEEE Softw. **19**(3), 22–24 (2002). https://doi.org/10.1109/MS.2002.1003449
12. Van Rompaey, B., Demeyer, S.: Establishing traceability links between unit test cases and units under test. In: 2009 13th European Conference on Software Maintenance and Reengineering, pp. 209–218. IEEE (2009)
13. Wang, S., Wen, M., Liu, Y., Wang, Y., Wu, R.: Understanding and facilitating the co-evolution of production and test code. In: 2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER), pp. 272–283. IEEE (2021)
14. White, R., Krinke, J., Tan, R.: Establishing multilevel test-to-code traceability links. In: Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering, pp. 861–872 (2020)
15. Zaidman, A., Rompaey, B., Deursen, A., Demeyer, S.: Studying the co-evolution of production and test code in open source and industrial developer test processes through repository mining. Empirical Softw. Eng. **16**, 325–364 (2011). https://doi.org/10.1007/s10664-010-9143-7
16. Zerouali, A., Mens, T.: Analyzing the evolution of testing library usage in open source java projects. In: 2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER), pp. 417–421. IEEE (2017)