Hybrid Test-Smell Based Approach for Prediction of Flaky Tests

Saurabh Bodke
sb5470@rit.edu

Follow this and additional works at: https://repository.rit.edu/theses

Recommended Citation

This Thesis is brought to you for free and open access by the RIT Libraries. For more information, please contact repository@rit.edu.
Hybrid Test-Smell Based Approach for Prediction of Flaky Tests

by

Saurabh Bodke

A thesis submitted in partial fulfillment of the requirements for the degree of

Master of Science
in Computing and Information Sciences

B. Thomas Golisano College of Computing and Information Sciences
Rochester Institute of Technology
Rochester, NY

December 12, 2023

Approved by:

Dr. Mohamed Wiem Mkaouer
Dr. Ali ben Mrad
Dr. Makram Soui
Abstract

Regression testing is an essential component in software development, aimed at verifying that recent code changes do not negatively impact existing functionalities. A prevalent challenge in this area is the occurrence of flaky tests, which exhibit inconsistent outcomes—passing or failing without any code alterations. Such tests erode the reliability of automated testing frameworks, leading to increased debugging efforts and diminished confidence in software stability.

This research addresses the challenge of flaky tests by exploring machine learning and prediction models, specifically focusing on test smells and four principal root causes of flakiness: Async Wait, Concurrency, Test Order Dependency, and Resource Leak. While existing methods that analyze test case vocabulary often encounter issues such as elevated operational costs, tendencies to overfit, and context sensitivity, our study enhances the accuracy of identifying flaky tests by incorporating these four additional determinants into the test smell-based approach.

We conducted a thorough evaluation of this augmented model’s predictive capabilities in a cross-project context and analyzed the contribution of each newly added flakiness factor in terms of information gain. Our findings indicate that Async Wait and Concurrency, in particular, show the highest information gain, highlighting their significant role in flakiness prediction. Moreover, we compared the performance of our hybrid feature based approach against both the vocabulary-based approach and the traditional test-smell based approach, focusing on the improved accuracy, precision, recall in predicting flaky tests.
I would like to dedicate this thesis to my family for their encouragement and support throughout my career. I appreciate all the love and support you have given me and this accomplishment is as much yours as much as mine.

With all my love and gratitude,

Saurabh Arun Bodke
Contents

1 Introduction 1

2 Background 4
  2.1 Regression Testing ................................. 4
  2.2 Flaky Tests ........................................ 5
  2.3 Test Smells ........................................ 5
  2.4 Async Wait ........................................ 6
  2.5 Concurrency ....................................... 8
  2.6 Test Order Dependency ............................ 9
  2.7 Resource leak ..................................... 10

3 Research Objective 13
  3.1 Motivation ......................................... 13
  3.2 Research Questions ............................... 14

4 Related Work 16

5 Methodology 19
  5.1 Data Preparation ................................. 19
CONTENTS

5.1.1 Data Scraping ................................................ 20
5.1.2 Pattern Identification ............................... 20
5.1.3 Data Merging ................................................ 21
5.2 Features .......................................................... 21
  5.2.1 Existing Features ........................................ 21
  5.2.2 New Features .............................................. 22
5.3 Used Classifiers .................................................. 23
5.4 Approach .......................................................... 23
5.5 Evaluated Metrics ............................................... 26

6 Analysis of Results .............................................. 28

7 Threats to Validity ............................................... 38
  7.1 Construct Validity ............................................. 38
  7.2 Internal Validity .............................................. 38
  7.3 External Validity .............................................. 39

8 Conclusion .......................................................... 40

9 Acknowledgement .................................................. 42
List of Tables

6.1 Hybrid Feature Based classifier’s performance . . . . . . . . . . 29
6.2 Test smells-based classifiers' performance. . . . . . . . . . . . . 30
6.3 Summary of information gain and occurrences of flaky tests. . . 35
6.4 Cross-project Hybrid Feature-Based classification . . . . . . . 36
6.5 Cross-project Test Smells-Based classification . . . . . . . . . 36
6.6 Cross-project Vocabulary-Based classification . . . . . . . . . 37
Chapter 1

Introduction

A critical technique used by software teams to maintain quality during software development is regression testing. This process involves re-evaluating the system’s quality by examining test results following code modifications. The successful results of the tests indicate a functioning system, while failures may signal a decline in quality.

A significant challenge arises when tests yield inconsistent results without any corresponding code changes, complicating the regression testing process for developers and quality assurance teams. These unpredictable tests, known as "flaky tests", produce arbitrarily varying results. This unpredictability frustrates developers, complicates the interpretation of test outcomes, and results in inefficient use of time and resources. Furthermore, flaky tests are challenging to rectify and may obscure deeper issues within the software’s design. Early identification and resolution of such tests are crucial for maintaining overall system integrity.

Recent research has increasingly focused on the challenge of identifying
CHAPTER 1. INTRODUCTION

flaky tests, with two primary methodologies emerging: Dynamic and Static approaches. The Dynamic approach involves repeatedly executing test cases to detect inconsistent results. Although effective, it can be resource-heavy and prone to errors, compounded by the difficulty in determining optimal test execution frequencies. Much research in this field employs machine learning (ML) techniques to estimate the likelihood of test case flakiness. The selection and analysis of specific ML features are critical to the uniqueness and success of each study.

In contrast, the Static approach analyzes test code for flakiness indicators without actual execution. This method efficiently identifies potential flakiness by examining code structure and patterns, offering a rapid and cost-effective alternative. However, its major drawback is the potential to miss flakiness aspects that manifest only during live test execution. Dynamic and Static approaches each have their merits and limitations. The Dynamic method provides direct observation of flakiness but is resource-intensive. The Static method is more efficient but may overlook certain flakiness elements that are only observable during execution. The choice between these methods depends on factors such as resource availability, time constraints, and specific software project requirements.

Pinto et al. [108] pioneered a static feature-based approach for flaky test identification, targeting automated detection through test code pattern analysis. Camara et al. [59] later extended Pinto et al.’s [108] work, evaluating its real-world applicability and cross-project performance. They identified a critical limitation in the vocabulary-based technique: susceptibility to context sensitivity and overfitting in diverse project settings.

To address these issues, Camara et al. [59] introduced a test smell-based
CHAPTER 1. INTRODUCTION

approach, relying solely on static metrics. This approach included test case size, test smell count, and binary indicators for 19 specific test smells, smells count, and LOC(Lines of code). Their findings suggested that the test-smell based approach outperformed the vocabulary-based approach in cross-project predictions. However, opportunities for enhancement were noted, especially with the inclusion of additional binary features.

Our paper builds upon Camara et al.’s [59] methodology, incorporating four new binary features representing primary flaky test root causes: Async Wait, Concurrency, Test Order Dependency, and Resource Leak. Luo et al. [78] identified these factors as significant contributors to the majority of flaky tests, affecting the quality of the Code Under Test (CUT). By expanding the feature set to 25, our study aims to further improve the predictive accuracy, precision, and recall for test flakiness. Utilizing the same dataset and classifiers as prior research, this paper evaluates the effectiveness of the hybrid feature based approach, striving to establish a new standard in flaky test prediction.
Chapter 2

Background

This section provides an overview of the main concepts critical to software testing, particularly in the realm of identifying and mitigating issues that can compromise software application integrity. Understanding these concepts is essential for comprehending the challenges in software testing and appreciating the methodologies and solutions proposed in this study.

2.1 Regression Testing

Regression testing ensures that recent code changes have not adversely affected existing features. It involves re-running functional and non-functional tests to verify that the behavior of the software remains consistent after modifications. For example, if a new feature is added to an email application, regression testing would confirm that existing functionalities like sending and receiving emails are still working as intended.
2.2 Flaky Tests

Flaky tests refer to tests that give inconsistent results, passing or failing intermittently without any changes to the code. These tests are problematic because they can lead to false positives or negatives, which undermines the reliability of the testing process. For instance, a flaky test in a web application might sometimes fail due to network latency issues, even though the application code is correct. We have shared code snippets of flaky tests further in Figure 2.2, Figure 2.3, and Figure 2.4 to describe the 4 primary root causes of Flaky tests i.e Async wait, Concurrency, Test Order Dependency, and Resource Leak as proposed by Luo et al. [78].

2.3 Test Smells

Test smells are patterns in the test code that suggest a potential issue, often indicating poor design or maintainability problems. They are analogous to code smells in production code.

Deursen et al. [68] initially identified several test smells, including Assertion Roulette, Eager Test, General Fixture, and Lazy Test, which highlighted prevalent issues in test programming. Building upon Deursen’s foundational work, Peruma et al. [97] further extended the scope of test smells. They incorporated additional categories inspired by prevalent shortcomings in unit test programming techniques, as documented in the existing literature. In subsequent research, Peruma and colleagues developed 'TSDetect', an open-source tool specifically designed for the detection of code smells. This tool, TSDetect, has been employed in the current study to identify test smells.
CHAPTER 2. BACKGROUND

6

Figure 2.1: Test smell example

The code snippet in Figure 2.1 shows a method from the test class EventsScraperTest.java, from the open source project TuCanMobile. This example is a flaky test extracted from the test smells examples mentioned on the testsmell.org website maintained by Peruma et al.

The test method, testSpinner(), contains multiple control statements (i.e. control flow statements). The success or failure of the test is based on the result of the assertion method which is within the control flow blocks and hence not predictable. This also increases the complexity of the test method and hence has a negative impact on maintenance of the test.

2.4 Async Wait

In the context of asynchronous programming, 'Async Wait' is a critical concept where the execution of a program is temporarily halted until a specified condition is fulfilled or an operation completes. In our context, it refers to a
category of tests where the tests on execution make an async call and do not adequately wait for the result of the call to become available before using it. Such type of test codes contributes majorly to flaky tests. For example, a test might fail intermittently if it does not correctly wait for an asynchronous API call to complete before asserting the outcome.

```java
@Test
public void testRasReportsWrongServerName() throws Exception {
    MiniHBaseCluster cluster = TEST_UTIL.getHBaseCluster();
    MiniHBaseClusterRegionServer firstServer = 
        (MiniHBaseClusterRegionServer)cluster.getRegionServer(0);
    HServerInfo hsi = firstServer.getServerInfo();
    firstServer.setHServerInfo(...);

    // Sleep while the region server pings back
    Thread.sleep(2000);
    assertTrue(firstServer.isOnline());
    assertEquals(2, cluster.getLiveRegionServerThreads().size());
    ... // similarly for secondServer
}
```

Figure 2.2: Async Wait flaky test

The code snippet in Figure 2.2 represents a Async Wait flaky test. This snippet is from the HBase project, used by Luo et al. [78] to demonstrate Async wait root cause in tests (or the CUT). The test relies on a fixed-duration `Thread.sleep(2000)` to wait for an asynchronous operation to complete before proceeding with assertions. The test assumes that two seconds is always sufficient time for the firstServer to initialize and respond ("ping back"). However, if the server takes longer than two seconds due to variability in network latency, load, or other environmental conditions, the assertions that follow the sleep will execute before the server is ready, likely causing the test to fail.

This kind of flakiness arises because the test’s success is contingent upon an asynchronous event—the server’s response—occurring within a predetermined
time window, which cannot be guaranteed under all execution circumstances. Thus, it falls into the async wait category of flaky tests, as the timing of the response and the test’s execution are not synchronized, leading to non-deterministic test outcomes.

2.5 Concurrency

Concurrency in software engineering is about components or processes executing independently in parallel, which can lead to complex states and behaviors in an application. For testing, concurrency issues might arise when multiple processes interact in unpredictable ways, causing flaky tests. Common problems include race conditions, where the test result depends on the order of parallel operations, other prevalent concurrency-related problems include atomicity violations, where intended atomic sequences of operations are interrupted by concurrent activities, and deadlocks, which occur when two or more processes are waiting indefinitely for each other to release resources. Tests with such root cause of flakiness falls under Concurrency category.

We next describe a code snippet of a Concurrency flaky test. This snippet

```java
1 if (conf != newConf) {
2     for (Map.Entry<String, String> entry : conf) {
3         if ((entry.getKey()).matches("hcat.*")) &&
4             (newConf.containsKey(entry.getKey())) == true) {
5             newConf.put(entry.getKey(), entry.getValue());
6         }
7     }
8     conf = newConf;
9 }
```

Figure 2.3: Concurrency flaky test

The code snippet in Figure 2.3 is from the Hive project, used by Luo et
al. [78] to demonstrate Concurrency root cause in the CUT. The code traverses a map that is accessed by multiple threads. Flaky test failures occur when these threads modify the map at the same time, resulting in a ConcurrentModificationException [60].

2.6 Test Order Dependency

Test order dependency occurs when the outcome of a test depends on the sequence in which tests are run, leading to inconsistent results. This typically happens when tests share the state between them. For instance, if a test for a user login feature always passes when run after a specific user creation test due to shared state, but fails when run independently, it exhibits a test order dependency.

```java
@BeforeClass
public static void beforeClass() throws Exception {
    bench = new TestDFSIO();
    ...;
    cluster = new MiniDFSCluster.Builder(...).build();
    FileSystem fs = cluster.getFileSystem();
    bench.createControlFile(fs, ...);
    // Check write here, as it is required for other tests */
    testWrite();
}
```

Figure 2.4: Test Order Dependency induced flaky test

The code snippet in the Figure Figure 2.4 is from the Hadoop project, used by Luo et al. [78] to demonstrate Test Order Dependency root cause in the CUT. The testWrite() method writes data to a file using fs, setting up the data for other tests to read. Initially, the developers assumed that testWrite() would always execute first. However, JUnit does not ensure a specific order of test
execution. Consequently, if JUnit runs any of the read tests before testWrite, the test fails due to the absence of the expected data. To resolve this, the developers modified their approach: they removed the standalone testWrite() test and incorporated a call to testWrite() within the beforeClass() method, as indicated in line 10. This change ensures that testWrite() executes once before any other tests in the class, preparing the necessary data setup for subsequent tests.

2.7 Resource leak

Resource leaks occur when an application fails to properly manage system resources, such as file handles or memory allocations. These leaks can cause flaky tests that pass or fail unpredictably, dependent on the availability and state of the resource at the time of test execution. The attached figure presents an illustrative example of a resource leak within a test case designed to assess resource allocation.

The code snippet in the example Figure 2.6 is used by Lam et al. [76], to demonstrate Resource leak root cause in the CUT. ResourceAllocation() tests whether the necessary resources are properly allocated for an application. The application internally uses a third-party database to store some information. To ensure isolation between test cases, the TestCleanup() method is designated to delete the database file, allowing for a fresh environment for each subsequent test. However, an issue arises due to the behavior of the third-party database library used by the application. This library does not immediately release the file handle upon request; it requires the garbage collector to execute before the file handle is released. Consequently, if the garbage collector has not run by
CHAPTER 2. BACKGROUND

1 DatabaseProvider ssp;
2 String currentDirectory = ...;
3 [TestMethod]
4 public void ResourceAllocation() {
5     ssp = new DatabaseProvider(currentDirectory);
6     ...
7 }
8 [TestCleanup]
9 public void TestCleanup() {
10     ClearConnections();
11     if (File.Exists(dbPath)) {
12         File.Delete(dbPath);
13     }
14     ...
15 }

Figure 2.5: Resource Leak induced flaky test

1 DatabaseProvider ssp;
2 String currentDirectory = ...;
3 [TestMethod]
4 public void ResourceAllocation() {
5     ssp = new DatabaseProvider(currentDirectory);
6     ...
7 }
8 [TestCleanup]
9 public void TestCleanup() {
10     ClearConnections();
11     if (File.Exists(dbPath)) {
12         File.Delete(dbPath);
13     }
14     ...
15 }

Figure 2.6: Resource Leak induced flaky test
the time File.Delete(dbPath) is invoked in Line 12, an exception is thrown due to the file still being in use, leading to a flaky test outcome
Chapter 3

Research Objective

3.1 Motivation

This research is primarily inspired by Camara et al.’s [59] exploration of a test smell-based approach for predicting test flakiness. Although this approach has shown promising results in cross-project validation, its performance in inter-project contexts is comparatively weaker, especially against the vocabulary-based approach, which is currently recognized as the state-of-the-art (SOTA) method. Furthermore, the precision of the test smell-based models is observed to be up to 14% lower than their vocabulary-based counterparts in optimal scenarios. In response to these findings, our research aims to improve the performance of the test smell-based model by integrating additional static features alongside test smells. Our goal is to achieve marked advancements in both inter- and intra-project contexts, positioning our approach as a more generalized solution for cross-project predictions.
3.2 Research Questions

We have defined three Research Questions, based on our research goals:

**RQ1:** *How does the predictive accuracy of the Hybrid Feature-Based approach, which combines test smells with additional code metrics, compare to the traditional Test-Smell-Only approach in predicting test flakiness?*

- The objective here is to evaluate the effectiveness of our model in terms of accuracy. By comparing it with the test-smell-only approach, we aim to validate the enhancements our method offers in predicting flakiness. Success will be measured by our model’s ability to exceed the accuracy of the test-smell-based approach.

**RQ2:** *How strongly are the new features associated with test flakiness prediction?*

- This question seeks to determine the impact and relevance of the new features in flakiness prediction. We plan to compute and analyze the information gain metrics for each new feature, providing insight into their individual contributions and their importance in improving the predictive accuracy of our model. Understanding the influence of each new feature on the model’s ability to accurately predict test flakiness is crucial.

**RQ3:** *How does the performance of the Hybrid Feature-Based ap-
proach, contrast with that of the established vocabulary-based and test-smell-only approaches in the prediction of test flakiness in the cross-project context?

- The aim here is to assess whether our approach outperforms the test-smell-based and vocabulary-based methods, particularly in a cross-project setting. This comparison is essential to establish the effectiveness of our approach as a more generalized solution in diverse application scenarios.
Chapter 4

Related Work

Many studies addressed challenges in software maintenance in general [2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99, 100, 101, 102, 103, 104, 105, 106, 107, 108, 109, 110, 111, 112, 113, 114, 115, 116, 117, 118, 120, 121, 122, 123, 124, 125], and test flakiness in particular. The issue of flaky tests has emerged as a significant concern in software engineering, primarily due to its adverse impact on developer productivity and the overall software development lifecycle. Flaky tests, characterized by their non-deterministic nature under the same conditions, undermine the reliability of test suites and lead to increased maintenance efforts. Developers often find themselves repeatedly re-running tests to distinguish between genuine failures and flakiness, a process that is not only time-consuming but also diverts attention from critical development tasks. This constant need for verification and debugging can significantly slow down the development process,
leading to delayed releases and increased costs. Furthermore, the presence of flaky tests can erode trust in the testing process itself, making it difficult for teams to rely on automated tests for continuous integration and deployment. These challenges explain the sustained interest and ongoing efforts within the research community to address flaky tests [126] [81] [95]. By developing more effective detection and mitigation strategies, researchers aim to enhance the efficiency and accuracy of testing processes, ultimately contributing to more stable software products and more productive development environments.

W. Lam, R. Oei, and A. Shi [77] [119] have proposed adopting dynamic approaches which involve running the test suite for a fixed number of times. However, it increases the cost of execution and for large organizations, this becomes a scaling issue.

According to Pinto et al. [108], it is possible to extract a vocabulary of patterns of words from the test code that can be used to determine whether a test is flaky or not. The authors developed a dataset of Java projects with test cases labeled as flaky and non-flaky and then used it to train and evaluate ML systems. Overall, all classifiers performed well, with SVM having the best recall and Random Forest having the best precision (0.99) and F1-Score (0.92). The top 20 features with the best information gain are also displayed in the study. The effectiveness of vocabulary-based strategy in Python projects was also the subject of several research [1] [73].

Camera et al. [60] replicated the study by Pinto et al. [108] study. They extended the work by using the trained classifiers to predict flaky tests using a different test dataset. The classifiers were used for prediction in two different contexts namely Inter and Intra project contexts. Among the trained classifiers, LDA achieved the best results with recall of 0.75 in intra-project
contexts and 0.45 in inter-project scenarios. The authors concluded that the vocabulary-based approach, while effective in some settings, is sensitive to context and has a tendency to overfit.

Alshammari et al. [53] developed an approach, FlakeFlagger, to predict test flakiness, incorporating both static and dynamic features like test smells, test coverage, and source code management. They used a dataset of 24 open-source Java projects to benchmark FlakeFlagger against both a vocabulary-based [108] and a combination of both. While recall rates were comparable across all three approaches (74%, 72%, and 74%), FlakeFlagger displayed a significant 49% improvement in precision over the vocabulary-based approach. The hybrid approach further improved precision by 6%. Interestingly, although FlakeFlagger includes its own expanded test smell detector, the analysis revealed a low correlation between these test smells and test flakiness.

Camera et al. [59] evaluated the performance of the traditional test smell-based approach for prediction of flakiness. It was observed that using test smells are potentially good predictors of flakiness. The authors also compared their results with the vocabulary-based approach. The test smell-based models had the precision(83%) which is 14% lower than the state-of-the-art vocabulary based approach(97%). In the cross-project validation, the test smell-based approach in general performed better in the intra- and inter-project contexts.

The test smell-based approach produced promising results and opened the possibility to further add more static or dynamic features to the test smells as indicators. We extended Camara et al. [59] study on using test smells and explored the possibility of extending the features for predicting flaky tests.
Chapter 5

Methodology

5.1 Data Preparation

We based our study on the dataset used in Camara et al.’s [59] work, which originally consisted of 2953 Flaky tests and 1400 Non-Flaky tests. This dataset included 1400 Flaky and 1377 Non-Flaky tests from the msr4flakiness dataset, and an additional 153 Flaky tests from the idFlakies dataset, both sourced from Pinto et al.’s [108] earlier work. To gather the necessary data, we cloned the GitHub repository "https://github.com/ncsu-swat/msr4flakiness", which contained testCase Class files from Camara et al.’s [59] base dataset. This repository is divided into three sections: i) Flaky testCase Class files from msr4flakiness, ii) Non-Flaky testCase Class files from msr4flakiness, and iii) Flaky testCase Class files from idFlakies.
5.1.1 Data Scraping

We extracted the test methods, classified as either Flaky or Non-Flaky, from the testCase Class files. We then stored the code snippets of these test methods separately for all three sections to maintain distinct datasets.

5.1.2 Pattern Identification

Informed by the findings of Luo et al. [78], we acknowledged Async Wait, Concurrency, and Test Order Dependency as the primary root causes of flaky tests. To these established categories, we introduced an additional parameter: Resource Leak. Based on the survey of existing researches on flakiness prediction by Parry et al. [96], resource leaks were a major contributing factor to test flakiness in a large-scale study of open-source Java projects. We developed regex patterns to identify test methods associated with these four root causes within

Figure 5.1: Target Distribution
the code snippets. This identification process was executed separately for flaky tests in 'msr4Flakiness', non-flaky tests in 'msr4Flakiness', and flaky tests in 'idFlakies', ensuring a comprehensive analysis across varying test scenarios.

5.1.3 Data Merging

The pattern identification process yielded four new feature columns: “AsyncWait”, “Concurrency”, “TestOrderDependency”, and “ResourceLeak”. We then merged these columns with the base dataset “Sampled.csv” to create an enriched dataset for our study, encompassing both existing and newly added features, served as the primary dataset for our research.

5.2 Features

Machine learning algorithms must be trained on a set of features in order to derive patterns and predict outputs from given inputs. New features were developed in order to improve these quality measurements.

5.2.1 Existing Features

The dataset we utilized already comprised 21 features. Of these, 19 are specific test smells associated with each test case, capturing a broad spectrum of potential issues within the test code. The remaining two features focus on code complexity: LOC (Lines of Code), which measures the length of the code, and Smell Count, which aggregates the total number of test smells present. These features provide a foundational understanding of the test environment and its inherent complexities. [Reference: Test Smells Table]
CHAPTER 5. METHODOLOGY

5.2.2 New Features

To enhance the predictive power of our model, we have introduced four new binary features, each addressing a critical root cause of test flakiness:

Async Wait

Asynchronous operations are prevalent in modern software development. Flakiness in tests due to asynchronous waits often stems from issues such as race conditions or improper handling of asynchronous callbacks. Identifying asynchronous waits is crucial, as they are prone to intermittent failures due to timing discrepancies that may vary across environments or runs.

Concurrency

Concurrency introduces complexity in testing due to potential issues like deadlocks, race conditions, and resource contention. Tests with concurrent execution can yield unpredictable results, leading to non-deterministic outcomes in a multi-threaded environment. Analyzing such tests is vital to understand concurrency-related issues.

Test Order Dependency

This dependency implies that a test’s outcome hinges on the sequence of test execution. Identifying tests affected by shared states or side effects from other tests is crucial, as these factors can cause unpredictability in test suites.
Resource Leak

Resource Leak is identified as the root cause of a flaky test when the test’s outcome is directly influenced by the application’s failure to properly handle resources, such as acquiring or releasing file locks. These new features, representing the main root causes of flakiness in test methods have been chosen to augment our dataset. The decision aims to significantly refine the ability of the model to predict flaky tests [96] [78].

5.3 Used Classifiers

We have used the visualization library such as Matplotlib for Exploratory Data Analysis. We used 8 classifiers available in the framework Scikit-learn: Random Forest, Decision Tree (DT), Naive Bayes, Support Vector Machine (SVM), Logistic Regression (LR), Linear Discriminant Analysis (LDA), K-Nearest Neighbour (KNN), and Perceptron. These models are imported using Python’s scikit-learn library.

5.4 Approach

Our approach closely aligns with the experimental setup used by Camera et al. [59]. In their study, Camera et al. [59] developed models using a data set from Pinto et al. [108] and conducted cross-project evaluations with a dataset from Lam et al. [77], which exclusively comprises Flaky tests from 72 different projects. These datasets, processed with the tsDetect tool, yielded 19 test smell features, along with two additional features: LOC (lines of code) and smells count (total number of test smells present). They generated two
Figure 5.2: Correlation between feature column
In our research, we build on the work of Camera et al. [59], by incorporating four new features into the existing data set. Our updated dataset includes 1377 Flaky and 1400 Non-Flaky samples from “msr4flakiness” and 153 Flaky samples from “idFlakies”, totaling 25 features (both existing and new). We utilized “msr4Flakiness” samples for training the models and “idFlakies” samples for cross-project validation.

Following dataset preparation, we employed Matplotlib for exploratory data analysis, visualizing the data to gain insights. The preprocessing involved converting all feature columns to integers and transforming the dataset into numeric form. This process was crucial for the model to effectively interpret

Figure 5.3: KDE plot of LOC(Line of Code)
the features and the ground truth. A correlation matrix was also visualized to identify the relationships between features.

We conducted hyperparameter tuning using GridSearchCV. This methodology, which aimed to optimize classifier parameters, involved systematic experimentation with various combinations and permutations to identify the optimal parameter set, thus enhancing performance while avoiding over-fitting and under-fitting.

A range of models were trained using the training set, selected based on their prevalent use in classification tasks and compatibility with the dataset’s characteristics. RandomForest demonstrated the most effective performance. Finally, we tested these models for cross-project validation, comparing our results with those of Camera et al. [59], and Pinto et al. [108].

5.5 Evaluated Metrics

To evaluate the effectiveness of our predictive models, we used a comprehensive suite of metrics, each offering a unique perspective on model performance.

**Precision:** This metric calculates the ratio of correctly predicted flaky tests to the total number of predicted flaky tests. High precision indicates a lower rate of false positives, signifying that fewer non-flaky tests are incorrectly labeled as flaky.

**Recall:** Also known as sensitivity, recall measures the ratio of correctly predicted flaky tests to the actual flaky tests within the dataset. It assesses the model’s ability to identify all relevant instances, reflecting its detection capability.

**F1-Score:** Representing the harmonic mean of precision and recall, the
F1-Score balances these two metrics. This is particularly useful in scenarios where the class distribution is imbalanced.

**MCC (Matthews correlation coefficient):** MCC provides a holistic measure of the model’s quality by considering true positives, false positives, true negatives, and false negatives. It is especially effective in evaluating binary classification problems.

**AUC (Area under the ROC curve):** The AUC measures the model’s ability to distinguish between classes. Higher values indicate better classification performance, showcasing the degree of separability the model achieves.

For a more detailed analysis, we employed tools like confusion matrices and ROC (Receiver Operating Characteristic) curves. These visual representations are crucial in understanding the balance between true positive and false positive rates, offering a nuanced view of model performance.

In our cross-project validation, particularly in the intra- and inter-project evaluations, we focused on the ’idFlakies’ dataset. Due to the absence of non-flaky test examples in this dataset, we primarily used recall as the evaluation metric. This focus ensures that our assessment is appropriately aligned with the dataset’s structure and offers meaningful insights into the model’s ability to accurately identify flaky tests.
Chapter 6

Analysis of Results
This section analyses the obtained results to answer the research questions.

RQ1: How does the predictive accuracy of the Hybrid Feature-Based approach, which combines test smells with additional code metrics, compare to the traditional Test-Smell-Only approach in predicting test flakiness?

The Hybrid Feature-Based Model, which incorporates additional code metrics alongside traditional test smell features, has demonstrated an enhancement in predicting test flakiness. When assessing the performance using precision, recall, and F1 score, the hybrid model shows [Table 6.1] improved metrics across all classifiers when compared to the Test-Smell-Only [Table 6.2] Approach. Notably, the precision and recall values for the hybrid model range from approximately 75% to 85%, reflecting an improvement from the 74% to 83% range exhibited by the Test-Smell-Only Approach. The Random Forest classifier, in particular, showed an increase from 0.83 to 0.85 in both precision and recall when using the hybrid model. Additionally, the hybrid model demonstrates a stronger correlation between the predicted and actual classifications.
Table 6.1: Hybrid Feature Based classifier’s performance

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Precision</th>
<th>Recall</th>
<th>F1</th>
<th>MCC</th>
<th>AUC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Random Forest</td>
<td>0.85</td>
<td>0.85</td>
<td>0.85</td>
<td>0.71</td>
<td>0.92</td>
</tr>
<tr>
<td>Decision Tree</td>
<td>0.84</td>
<td>0.84</td>
<td>0.84</td>
<td>0.68</td>
<td>0.87</td>
</tr>
<tr>
<td>KNN</td>
<td>0.81</td>
<td>0.81</td>
<td>0.81</td>
<td>0.63</td>
<td>0.87</td>
</tr>
<tr>
<td>LR</td>
<td>0.81</td>
<td>0.81</td>
<td>0.81</td>
<td>0.62</td>
<td>0.87</td>
</tr>
<tr>
<td>Perceptron</td>
<td>0.81</td>
<td>0.81</td>
<td>0.81</td>
<td>0.62</td>
<td>N/A</td>
</tr>
<tr>
<td>LDA</td>
<td>0.80</td>
<td>0.80</td>
<td>0.79</td>
<td>0.60</td>
<td>0.87</td>
</tr>
<tr>
<td>SVM</td>
<td>0.75</td>
<td>0.75</td>
<td>0.75</td>
<td>0.51</td>
<td>0.84</td>
</tr>
<tr>
<td>Naive Bayes</td>
<td>0.75</td>
<td>0.70</td>
<td>0.68</td>
<td>0.45</td>
<td>0.83</td>
</tr>
</tbody>
</table>

for flakiness, with MCC values exceeding 0.6 for six out of the eight classifiers, compared to the Test-Smell-Only Approach, which has three classifiers above this threshold. In comparison, the Traditional test-smell based Model had lower MCC values, with the Naive Bayes classifier scoring as low as 0.37 and the hybrid-feature based model having a higher MCC value of 0.71 for Random Forest classifier. In Figure 6.1 we can see the ROC curve and Precision-Recall curve and Confusion Matrix in Figure 6.2 for the Random Forest classifier in Hybrid approach.

Answer: These findings indicate that the inclusion of additional features related to the root causes of flakiness contributes significantly to the accuracy, precision, and recall of the predictive model. The
Table 6.2: Test smells-based classifiers’ performance.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Precision</th>
<th>Recall</th>
<th>F1</th>
<th>MCC</th>
<th>AUC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Random Forest</td>
<td>0.83</td>
<td>0.83</td>
<td>0.83</td>
<td>0.65</td>
<td>0.90</td>
</tr>
<tr>
<td>Decision Tree</td>
<td>0.83</td>
<td>0.83</td>
<td>0.83</td>
<td>0.66</td>
<td>0.86</td>
</tr>
<tr>
<td>KNN</td>
<td>0.81</td>
<td>0.81</td>
<td>0.81</td>
<td>0.62</td>
<td>0.81</td>
</tr>
<tr>
<td>LR</td>
<td>0.79</td>
<td>0.79</td>
<td>0.79</td>
<td>0.59</td>
<td>0.87</td>
</tr>
<tr>
<td>LDA</td>
<td>0.78</td>
<td>0.78</td>
<td>0.78</td>
<td>0.56</td>
<td>0.86</td>
</tr>
<tr>
<td>Perceptron</td>
<td>0.78</td>
<td>0.78</td>
<td>0.78</td>
<td>0.55</td>
<td>0.86</td>
</tr>
<tr>
<td>SVM</td>
<td>0.75</td>
<td>0.75</td>
<td>0.75</td>
<td>0.50</td>
<td>0.83</td>
</tr>
<tr>
<td>Naive Bayes</td>
<td>0.74</td>
<td>0.65</td>
<td>0.61</td>
<td>0.37</td>
<td>0.78</td>
</tr>
</tbody>
</table>

Figure 6.1: ROC and Precision-Recall Curve for RandomForest
Hybrid Feature-Based Model thus outperforms the Test-Smell-Only Approach, making it a more reliable and accurate tool for identifying flaky tests in software development.

Figure 6.2: Confusion Matrix for RandomForest

RQ2: How strongly are the new features associated with test flakiness prediction?

The calculation of information gain is a reliable method for evaluating the
importance of a feature in making predictions. It is based on the reduction of entropy, which measures how a feature can simplify the classification process. From the [Table 6.3], we see that loc (code lines) has the highest information gain, indicating that it is a significant predictor of flakiness in the context of this dataset. The next most informative feature is 'assertionRoulette', followed by 'Concurrency', 'smellsCount', and 'AsyncWait'. These features appear to have a strong association with test flakiness, as evidenced by their information gain. In Figure 6.3 we see the top 5 features with the most information gain.

Furthermore, when looking at the proportion of affected tests that are actually flaky, 'Concurrency' has a high percentage of flaky occurrences (87.57%), followed closely by 'Async Wait' (87.76%), and 'Resource Leak' stands out with the highest at 92%. This high prevalence of flaky occurrences for these new features demonstrates their strong correlation with flaky tests.

![Top 5 Features by Information Gain](image)

Figure 6.3: Top 5 information gain)

**Answer:** In summary, the new features 'Async Wait', 'Concurrency',
and 'Resource Leak' have a strong association with test flakiness prediction, as evidenced by their high information gain values and the significant proportion of flaky tests they affect. These features’ addition to the predictive model is justified by their substantial contribution to identifying flaky tests, enhancing the model’s performance.

RQ3: How does the performance of the Hybrid Feature-Based approach, contrast with that of the established vocabulary-based and test-smell-only approaches in the prediction of test flakiness in the cross-project context?

**Intra-Project Context:** The Hybrid Feature-Based approach in [Table 6.4] demonstrates superior performance compared to the test-smell-only approach in [Table 6.5], with three classifiers achieving recall values greater than 70%. Specifically, Logistic Regression and Perceptron both show a recall value of 0.74. Compared to the vocabulary-based approach in [Table 6.6], the Hybrid Feature-Based approach significantly outperforms the best classifier of the vocabulary-based approach, with KNN achieving a recall value of 57% against the 74% recall of Logistic Regression and Perceptron in the hybrid model.

**Inter-Project Context:** The test-smell-only approach in [Table 6.5] slightly outperforms the Hybrid Feature-Based approach in [Table 6.4], with SVM achieving a recall of 0.55 in the test-smell-only approach compared to a 0.51 recall in the hybrid model. When compared to the vocabulary-based approach in [Table 6.6], the best classifier of the Hybrid Feature-Based approach, SVM, has a lower recall value of 51% versus the 56% achieved by LDA in the vocabulary-based approach. However, considering all classifiers, the Hybrid Feature-Based
In summary, the Hybrid Feature-Based approach excels in the intra-project context, surpassing both the test-smell-only and the vocabulary-based approaches in terms of recall. In the inter-project context, while the hybrid approach’s best classifier has a lower recall value compared to the test-smell-only and vocabulary-based approaches, its overall performance across all classifiers suggests a generally better performance in cross-project validation. This indicates that the Hybrid Feature-Based approach has enhanced predictive capabilities in the intra-project setting and suggests the potential for further optimization in the inter-project context to achieve better generalization.
### CHAPTER 6. ANALYSIS OF RESULTS

<table>
<thead>
<tr>
<th>Pos.</th>
<th>Features</th>
<th>Inf. Gain</th>
<th>Total</th>
<th>Flaky</th>
<th>% Flaky</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>LOC</td>
<td>$0.2548171e-01$</td>
<td>2775</td>
<td>1377</td>
<td>49.58%</td>
</tr>
<tr>
<td>1</td>
<td>Assertion Roulette</td>
<td>$0.832619e-02$</td>
<td>1388</td>
<td>968</td>
<td>69.74%</td>
</tr>
<tr>
<td>2</td>
<td>Concurrency</td>
<td>$0.470961e-02$</td>
<td>314</td>
<td>275</td>
<td>87.58%</td>
</tr>
<tr>
<td>3</td>
<td>Smells Count</td>
<td>$0.270147e-02$</td>
<td>2653</td>
<td>1356</td>
<td>51.11%</td>
</tr>
<tr>
<td>4</td>
<td>Async Wait</td>
<td>$0.236097e-02$</td>
<td>188</td>
<td>165</td>
<td>87.77%</td>
</tr>
<tr>
<td>5</td>
<td>Sleepy Test</td>
<td>$0.194695e-02$</td>
<td>112</td>
<td>105</td>
<td>93.75%</td>
</tr>
<tr>
<td>6</td>
<td>General Fixture</td>
<td>$0.160650e-02$</td>
<td>267</td>
<td>61</td>
<td>22.85%</td>
</tr>
<tr>
<td>7</td>
<td>Duplicate Assert</td>
<td>$0.154864e-02$</td>
<td>376</td>
<td>269</td>
<td>71.54%</td>
</tr>
<tr>
<td>8</td>
<td>Constructor Initialization</td>
<td>$0.103536e-02$</td>
<td>68</td>
<td>63</td>
<td>92.65%</td>
</tr>
<tr>
<td>9</td>
<td>Print Statement</td>
<td>$0.105576e-02$</td>
<td>58</td>
<td>55</td>
<td>94.83%</td>
</tr>
<tr>
<td>10</td>
<td>Sensitive Equality</td>
<td>$0.840032e-03$</td>
<td>129</td>
<td>95</td>
<td>73.64%</td>
</tr>
<tr>
<td>11</td>
<td>Lazy Test</td>
<td>$0.440699e-03$</td>
<td>1786</td>
<td>817</td>
<td>45.74%</td>
</tr>
<tr>
<td>12</td>
<td>Resource Optimism</td>
<td>$0.426438e-03$</td>
<td>75</td>
<td>17</td>
<td>22.67%</td>
</tr>
<tr>
<td>13</td>
<td>Conditional Test Logic</td>
<td>$0.419574e-03$</td>
<td>356</td>
<td>219</td>
<td>61.52%</td>
</tr>
<tr>
<td>14</td>
<td>Resource Leak</td>
<td>$0.320018e-03$</td>
<td>25</td>
<td>23</td>
<td>92.00%</td>
</tr>
<tr>
<td>15</td>
<td>Unknown Test</td>
<td>$0.213505e-03$</td>
<td>544</td>
<td>234</td>
<td>43.01%</td>
</tr>
<tr>
<td>16</td>
<td>Verbose Test</td>
<td>$0.177088e-03$</td>
<td>7</td>
<td>7</td>
<td>100.00%</td>
</tr>
<tr>
<td>17</td>
<td>Magic Number Test</td>
<td>$0.109571e-03$</td>
<td>411</td>
<td>227</td>
<td>55.23%</td>
</tr>
<tr>
<td>18</td>
<td>Mystery Guest</td>
<td>$0.547207e-04$</td>
<td>124</td>
<td>71</td>
<td>57.26%</td>
</tr>
<tr>
<td>19</td>
<td>Eager Test</td>
<td>$0.245867e-04$</td>
<td>970</td>
<td>496</td>
<td>51.13%</td>
</tr>
<tr>
<td>20</td>
<td>Redundant Assertion</td>
<td>$0.827896e-03$</td>
<td>8</td>
<td>4</td>
<td>50.00%</td>
</tr>
<tr>
<td>21</td>
<td>Default Test</td>
<td>$0.000000e+00$</td>
<td>0</td>
<td>0</td>
<td>0.00%</td>
</tr>
<tr>
<td>22</td>
<td>Empty Test</td>
<td>$0.000000e+00$</td>
<td>0</td>
<td>0</td>
<td>0.00%</td>
</tr>
<tr>
<td>23</td>
<td>Ignored Test</td>
<td>$0.000000e+00$</td>
<td>0</td>
<td>0</td>
<td>0.00%</td>
</tr>
<tr>
<td>24</td>
<td>Test Order Dependency</td>
<td>$0.000000e+00$</td>
<td>0</td>
<td>0</td>
<td>0.00%</td>
</tr>
</tbody>
</table>

Table 6.3: Summary of information gain and occurrences of flaky tests.
### Table 6.4: Cross-project Hybrid Feature-Based classification

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Intra-Project</th>
<th>Inter-Project</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Recall</td>
<td>TP</td>
</tr>
<tr>
<td>Random Forest</td>
<td>0.71</td>
<td>25</td>
</tr>
<tr>
<td>Decision Tree</td>
<td>0.65</td>
<td>23</td>
</tr>
<tr>
<td>KNN</td>
<td>0.48</td>
<td>17</td>
</tr>
<tr>
<td><strong>LR</strong></td>
<td>0.74</td>
<td>26</td>
</tr>
<tr>
<td><strong>Perceptron</strong></td>
<td>0.74</td>
<td>26</td>
</tr>
<tr>
<td>LDA</td>
<td>0.65</td>
<td>23</td>
</tr>
<tr>
<td><strong>SVM</strong></td>
<td>0.66</td>
<td>23</td>
</tr>
<tr>
<td>Naive Bayes</td>
<td>0.57</td>
<td>20</td>
</tr>
</tbody>
</table>

### Table 6.5: Cross-project Test Smells-Based classification

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Intra-Project</th>
<th>Inter-Project</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Recall</td>
<td>TP</td>
</tr>
<tr>
<td>Random Forest</td>
<td>0.69</td>
<td>24</td>
</tr>
<tr>
<td>Decision Tree</td>
<td>0.66</td>
<td>23</td>
</tr>
<tr>
<td>KNN</td>
<td>0.51</td>
<td>18</td>
</tr>
<tr>
<td><strong>LR</strong></td>
<td>0.74</td>
<td>26</td>
</tr>
<tr>
<td>LDA</td>
<td>0.66</td>
<td>23</td>
</tr>
<tr>
<td>Perceptron</td>
<td>0.71</td>
<td>25</td>
</tr>
<tr>
<td><strong>SVM</strong></td>
<td>0.66</td>
<td>23</td>
</tr>
<tr>
<td>Naive Bayes</td>
<td>0.57</td>
<td>20</td>
</tr>
</tbody>
</table>
### Table 6.6: Cross-project Vocabulary-Based classification

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Intra-Project</th>
<th>Inter-Project</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Recall</td>
<td>TP</td>
</tr>
<tr>
<td>Decision Tree</td>
<td>0.31</td>
<td>11</td>
</tr>
<tr>
<td>LDA</td>
<td>0.29</td>
<td>10</td>
</tr>
<tr>
<td>LR</td>
<td>0.20</td>
<td>7</td>
</tr>
<tr>
<td>Random Forest</td>
<td>0.17</td>
<td>6</td>
</tr>
<tr>
<td>Naive Bayes</td>
<td>0.17</td>
<td>6</td>
</tr>
<tr>
<td>SVM</td>
<td>0.09</td>
<td>3</td>
</tr>
<tr>
<td><strong>KNN</strong></td>
<td><strong>0.57</strong></td>
<td><strong>20</strong></td>
</tr>
<tr>
<td>Perceptron</td>
<td>0.34</td>
<td>12</td>
</tr>
</tbody>
</table>
Chapter 7

Threats to Validity

7.1 Construct Validity

There may be instances where the tsDetect tool fails to correctly identify the production class during the preprocessing of the test code. This misidentification can lead to missing test smells, potentially compromising the validity of the outcome. Ensuring accurate detection of production classes is critical to the reliability of test smell extraction.

7.2 Internal Validity

The relationship between independent and dependent variables must be carefully managed to avoid skewing the results. Specifically, the absence of non-flaky classes in the cross-project dataset limits our ability to compute precision and other metrics, which could serve as benchmarks for the study. The inability to measure these metrics may impact the study’s findings and their implications.
7.3 External validity

Since our study focuses exclusively on Java and encompasses a select set of project domains, generalizing the results beyond this scope is not feasible. The results may not be representative of other programming languages or project domains. Additionally, a significant difference in the size of the datasets between the Intra-project and cross-project datasets could lead to discrepancies in model performance. The study's findings should be interpreted with caution, considering these potential variations in dataset size and diversity.
Chapter 8

Conclusion

Regression testing is essential in the continuous delivery of high-quality software, and the challenge posed by flaky tests significantly impacts development processes and the quality of software products. Our study explores the use of an extended set of features, including test smells, as predictors of flakiness in tests. Through meticulous research and analysis, and by applying standard evaluation metrics, we have evaluated the effectiveness of these extended features in predicting flaky tests. Our experimental results reveal that models trained to identify flaky root causes, such as Async Wait and Concurrency, substantially improve the prediction of flaky tests. We have benchmarked this approach against both the test-smell and vocabulary-based methods in a cross-project context, aiming to establish a more generalized approach for cross-project validation.

Our research indicates that Async Wait and Concurrency are indeed valuable predictors of test flakiness, corroborated by the information gain outcomes of these flaky root causes. Notably, we observed a 3% increase in the precision,
and recall rate for the hybrid feature based approach. To further refine this approach, we see an opportunity in expanding the training dataset with more projects, which could improve the accuracy of our predictive model.

Although our study focuses on static methods, there is a promising avenue for future research to integrate dynamic approaches with our static method. Such a combination could provide a more comprehensive understanding of test flakiness and its predictors, leading to more robust and reliable software development practices.
Chapter 9

Acknowledgement

I extend my deepest gratitude to my advisor, Dr. Mohamed Wiem Mkaouer, for his invaluable guidance, support, and mentorship throughout the journey of my thesis research and defense.

This thesis would not have reached its fruition without his guidance and persistent encouragement. His approachable nature and readiness to share his knowledge made the complex process of thesis development more manageable and enlightening.

Thank you, Dr. Mkaouer, for being more than an advisor; a mentor, and a significant pillar of support in this academic endeavor.

Saurabh Arun Bodke
Bibliography


BIBLIOGRAPHY


[59] Bruno Camara, Marco Silva, Andre Endo, and Silvia Vergilio. On the use of test smells for prediction of flaky tests. In *Proceedings of the


[71] Sirine Gharbi, Mohamed Wiem Mkaouer, Ilyes Jenhani, and Montassar Ben Messaoud. On the classification of software change messages


[88] Mohamed Wiem Mkaouer, Marouane Kessentini, Slim Bechikh, Mel Ó Cinnéide, and Kalyanmoy Deb. On the use of many quality at-


