Engineering Accessible Software

Thesis proposal

Arun Krishna Vajjala
Department of Computer Science
George Mason University
Fairfax, VA 22030
akrishn@gmu.edu

Committee
Kevin Moran, University of Central Florida (Chair)
Brittany Johnson-Matthews, George Mason University
Andrian Marcus, George Mason University
Thomas LaToza, George Mason University
Vivian Motti, George Mason University

Abstract

Continuous integration (**CI**) is a principle in current software development focusing on enhancing software quality by detecting bugs earlier. A primary task in **CI** is executing the test suite to ensure software correctness, particularly after code modifications. The expected outcome of these tests should be deterministic. However, certain tests may exhibit non-deterministic behavior even when run on an unchanged codebase. Such non-deterministic tests, known as *Flaky Tests*, can pass or fail for the same version of the software. This inconsistent outcome reduces reliability and can complicate the **CI** process.

The traditional way to detect flaky tests is to rerun them multiple times and if a test produces both passing and failing outcomes without any changes in the codebase, it is confirmed flaky [**?**]. Rerunning tests can be costly, leading to unacceptable overhead expenses, and it may not detect flakiness in tests that behave inconsistently in different running environments. Hence, researchers have been exploring alternative methods to detect flaky tests effectively rather than rerunning them.

To address the issue of flaky tests, I began by conducting a rerun experiment, aiming to understand the limitations of this technique beyond its obvious costs. From the data gathered during this experiment, I detect a set of flaky tests. Using this dataset, I developed a machine learning classifier, named FlakeFlagger, designed to predict if a test is flaky or not. The goal was to use FlakeFlagger without the need for reruns by finding the similar symptoms a test shares with other identified flaky tests. I evaluated the performance of FlakeFlagger against state-of-the-art tools to gain a better understanding of its effectiveness.

Flaky tests could exist in their test suites even after being confirmed as flaky for various reasons (e.g. helpful to detect defects). Identifying which failure from these tests are flaky or not is another main challenges. Hence, I proposed machine learning and non machine learning approaches to identify which failure is flaky or not based on the tests failure logs. Using the failure logs for each test, I am currently leveraging them to identify the root causes of test flakiness. This helps understanding the flaky failures and determining how to address them.

Contents

List of Figures

List of Tables

1 Introduction

Software has become an integral part of everyone's lives and its impact continues to grow. Software takes many forms such as web applications, smartphone applications, and desktop or operating system applications, etc [?]. Our everyday lives depend on the use of critical software applications which allow us to bank, invest, get news, and communicate with others. Touch screen devices such as smartphones and tablets provide a quick and easy means of access to important information and functions within our daily lives. The abundance of critical software introduces the responsibility to ensure people of all abilities and skill level are able to use and access their information. Software is ideally designed as all encompassing, where any user can use it as intended under their own abilities, but that is far from the current situation of software.

Software accessibility has become more important as more users are dependent on smartphones and computers. People of different abilities have found it difficult to use software the way it is currently developed and designed [?]. According to the world health organization (WHO), 15% of people have some disability [?], making software accessibility more important to ensure all users are able to use applications as intended.Though software engineers and companies are ethically motivated to create more accessible software, the United States Government is also making efforts to require public websites and services to be accessible [?]. The government in conjunction with the American with Disabilities Act (ADA) introduced legislation which ”prohibits discrimination on the basis of disability in the activities of public accommodations” [?, ?]. This has lead to a 180% increase in more accessible software as of 2018 [?]. This change in policy increases a need for more accessible software and tools that will help developers make their applications more accessible.

This effort to make more accessible software, however, has found its limitations. Google and Apple have the largest distribution for applications for the market [?]. Google's Google Play and Apple's App Store make it convenient for users to both download and create their own applications [?]. Though, these companies have worked to make their own devices more accessible, most of the applications on their app stores are not controlled and developed by them, therefore being largely inaccessible [?]. Large corporations have made their guidelines available to developers to follow [?, ?], but prior work has suggested that, although there is an abundance of accessibility research and guidelines, there has not been much research or work done to educate the large community of developers on accessibility related issues [?, ?]. The current state of software accessibility tools are generally ignored by developers because of a lack of concise warnings and difficulty of use [?, ?].As a result, 95% of the android applications randomly mined had elements that violated android GUI accessibility guidelines making the applications less accessible [?]. This raises the need for better accessibility focused developer tools so that developers can make more accessible applications.

Software engineering research is constantly innovating how software is made. Software testing and developer tools are constantly evolving and are making their way into the accessibility space. Software testing has been around for decades, but has recently grown into a more complex field with the use of computer vision and machine learning techniques to automatically generate and run tests. Moran et al. [?], for example, created a testing framework that took in application screenshots and automatically reported GUI design violations set by Android, therefore, letting developers know that their applications are not in accordance with the guidelines that are suggested. This exciting new way of testing could provide an ample amount of ways to test for accessibility guideline violations without developers needing to explicitly check for violations on their own. An extremely important part of software engineering research is the constant need for new, robust developer tools to assist developers in the process of designing and creating software. These tools allow developers to have quick access to information and functions that make the development process more efficient. Zhang et al. [**?**] created a tool for developers to automatically label the elements in their code that helps screen readers identify each element on the screen. Simple, but intelligent tools such as the ones Zhang et al. [**?**] proposed would drastically improve how accessible software is made by providing an effortless way for developers to quickly incorporate accessibility features into their software.

Given the current state of HCI, accessibility, and software engineering research, we can work towards the intersection of these areas and develop exciting new ways to make accessible software. Using data, accessibility research, and software engineering tools, we can address the lack of software accessibility at its core by providing intelligent tools for developers to create accessible software in the developmental and design stages.

The thesis proposal is organized as follows: Section 2 provides an introduction of works towards the problem of test flakiness. The main contributions of the thesis are discussed in Section 3. Section 4 provides a summary of the findings related to the detection of flaky tests. Section 5 emphasizes current research on how to identify flaky *failures* and explores techniques for their detection. The main key points for my future work, which form the remainder of my PhD, are discussed in Section 6. Lastly, the current plan for the remaining phase of my PhD is outlined in Section 7.

2 Background

Test flakiness is a timely and relevant research problem in software testing. The first empirical study focusing on defining flaky tests was conducted by Luo et al. [**?**]. They investigate that of 201 historical commits, 51 open-source projects contain logs of fixed flaky tests. They classify these flaky tests into different categories such as Async Wait and Test Order dependent [**?**].

Flaky tests have various impacts on the process of software development at the developer, team, and organizational levels. The degree to which both awareness of flaky test results and the measurement of flaky test side-effects can greatly aid developers in making the right decision when fixing an issue. Eck et al. [**?**] show that the majority of developers who participate in their study face flaky tests yearly (109 out of 120) and 58% out of 109 deal with flaky tests on a monthly basis. Eck et al. [**?**] also show that 79% of 109 participants consider flaky tests as a moderate-to-serious problem and that 77% believe flaky tests are a time-consuming problem as they are hard to reproduce. Developers who encounter flaky tests have different opinions regarding the reliability of the test suite. Based on the study, 77% of developers consider the test suite to be not fully reliable if it contains at least one flaky test.

Flaky tests are hard to reproduce which make them difficult for developers to debug. Some characteristics of software environments such as operating systems (where a test suite has been executed) may play a critical role to either hide or reveal a flaky test. Out of 311 flaky tests from the dataset of flaky tests Lam et al. [**?**] provide, 97 flaky tests could not be reproduced locally with 100 runs. This emphasizes the *difficulty* of establishing a particular number of runs in order to observe flaky tests. If a flaky test has been detected after n-number of runs on-server, then how is a developer to determine the number of runs to observe the same flaky test in different environments? Also, this implies that most studies that limit their runs may not have observed all of the possible flaky test scenarios in their test suites. This demonstrates how observing flaky tests can be extremely challenging.

With the effort to fix flaky tests, do all flaky tests be fixed before releasing the software? There are many implications of why some of the already-detected flaky tests still exist in further software releases. First, developers may not be able to resolve the flaky test because the current resources are not sufficient enough to fix all detected flaky tests or that developers need to involve many decisions at the team or organization levels such as providing external resources. The second implication is related to how developers evalxuate their proposed fixes. For example, developers usually use the rerun technique to see if their changes eliminate flakiness in a test suite. However, the rerun technique is not guaranteed to confirm if a test is not flaky. It is also true that a test may flake due to various reasons in the future and some of them may not be observed yet. Another explanation is that developers could be aware of the side-effects of a specific flaky test and decide to keep it after developers measure the cost of fixing the flaky test. The worst implication that a developer can make is to just ignore the flaky test without any potential work toward fixing or analyzing it. Throve et al. [**?**] report that 13% of the total number of commits that deal with flaky tests do so by just skipping them or removing the tests that cause flakiness.

2.1 Detecting Test Flakiness

There are significant studies proposing tools to help developers detect flaky tests without manual inspections. Each of these tools has its own mechanism to detect flaky tests. This section presents some of these effective tools.

DeFlaker. Bell et al. [**?**] propose *DeFlaker*, a new approach to detect flaky tests from the first failure without the need to rerun. Their approach uses execution coverage to detect flaky tests. Specifically, if a test fails but does not cover any changed code, the tool labels the test as flaky. To accomplish this, their tool consists of three main phases. First, *DeFlaker* uses a syntactic diff from a version control system to determine which changes the tool needs to track. Then, *DeFlaker* starts to collect coverage of each change identified from the previous phase. In the final stage, test outcomes and coverage information are analyzed to determine if a test is flaky or not. Detecting flaky tests by *DeFlaker* requires collecting complete statement coverage for each line of code. Bell et al. [**?**] believe that collecting coverage for all lines of code can be expensive. In order to make their tool light-weighted, they designed the tool to collect differential coverage, which means collecting only the coverage of the changed lines instead of all lines of code. Bell et al. [**?**] also consider that syntactic change information may be insufficient and there is a need to monitor even some syntactically unchanged lines. For instance, if there are changes made to the inheritance structure of a class or method, this may have different dynamic results. Bell et al. [**?**] have evaluated their tool on 5,966 builds of 26 open-source projects. They have found 87 previously unknown flaky tests in 10 of these projects. They also show that *DeFlaker* was able to detect 1,874 flaky tests from 4,846 previously known failures, with a low false alarm rate (1.5%). Their evaluation includes a strong comparison of the rerun experiment which was a main contribution in their work. The evaluation also considers the performance (overhead of running) of their tool by showing that *DeFlaker* was very fast, with an average slowdown of only 4.6% across all of selected projects.

iDFlakies. Lam et al. [**?**] provide a framework called *iDFlakies* to detect and partially classify flaky tests. Their framework relies on rerun by reordering tests during their executions. With a time limit for rerunning certain test suites, *iDFlakies* runs a test suite based on the original order and with reordering approaches. If a test passes and fails within the same order of tests, *iDFlakies* labels this test as a non-order dependency (NOD) flaky test. The second case is when a test fails during reordering the tests while it passes with the original order. In this case, *iDFlakies* labels this test as an order-dependency (OD) flaky test. In the process of changing the order of tests, *iDFlakies* shows that NOD flaky tests can be detected in the way as they may keep their original orders e.g. reorder test #6 with #7 while all tests from #1 to #5 have the same orders and test #3 for example passes and fails. *iDFlakies* can only partially classify the flaky tests to OD and NOD and it does not have any further classification for NOD flaky tests. Lam et al. [**?**] mention that *iDFlakies* does not randomly reorder tests during their executions. There are four different configurations of rerun tests in addition to rerunning based on the original order of tests. The first configuration is called random-class, where *iDFlakies* runs test classes in random order but it keeps the order of methods in each class in their original order. Random-class-method is another configuration by hierarchically randomizing the order of the test classes first and then the methods within test classes without interleaving methods from different classes. The third one is called reverse-class where all classes are run in reverse orders and keep the methods in each class with their original order. The last one is called the reverse-class-method which reverses the order of all test classes and methods from the original order. The tool is configured to run up to a limit number of rounds (the amount of reruns needed), run time (how long developers can rerun each project), or based on the minimum of both. The depth-first technique is used when, for example, a number of rounds x is given, *iDFlakies* runs on the first module up to x-times before rerunning another module. Their evaluation has been based on Java projects which build with Maven and use JUnit. As a result of applying *iDFlakies* on 82 different projects (111 modules in total), 422 flaky tests have been detected. In detail, 213 (50.5%) are classified as OD flaky tests and 209 (49.5%) as NOD flaky tests. Lam et al. [**?**] find that the random-class-method is the most effective configuration to detect OD flaky tests. In their studies, they used a time limit for rerun different ordering of configurations by 57 hours per project.

The Vocabulary of Flaky Tests. Pinto et al. [**?**] apply machine learning algorithms in order to find flaky test "vocabularies" that aid in predicting new flaky tests. The approach is defined as a classification problem. This means that supervised learning algorithms are needed to have prior knowledge of some existing flaky and non flaky tests (known as a training dataset) to be able to predict the status of flakiness for unseen tests (known as a testing dataset). Collecting the vocabulary list is done by applying text analysis on the syntax of given flaky and non-flaky tests. In other words, test contents are processed through natural language processing (NLP) techniques, including identifier splitting, stemming, and stop word removal, to turn the content of each test to a list of vocabulary (called tokens). In addition to collecting tokens, this approach collects Java keywords from each test and considers the length of tests (line of codes) as extra details to help the classifier distinguish between flaky and non-flaky tests. This approach simply turns each test to a list of tokens, Java keywords, and a variable called test length. Pinto et al. [**?**] aim to measure how well a flaky test classifier can learn from test contents. This approach has been evaluated based on the F1-score (measuring the accuracy of prediction performance based on the recall and precision scores) and information gain (known as the usefulness of a single token/Java keyword in terms of helping the classifier to identify a flaky test). By applying multiple supervised learning algorithms, these metrics can show the strength of flaky test predictors. This approach uses the dataset provided by Bell et al. [**?**] as a source to evaluate their approach. Pinto et al. [**?**] find that their approach was effective to distinguish between flaky and non-flaky tests by achieving an F1-score of 0.95 for the identification of flaky tests. They find that the Random Forest (RF) is the most effective algorithm used toward this classification problem. They used different outcomes of processing NLP techniques as an input of the classifier. For example, they measure the effectiveness of using only Java keywords in the learning phase or using the whole set of tokens without stop words removals.

3 Thesis

3.1 Problem Statement

Test flakiness presents significant challenges in software testing and development. As previously discussed in Section 2, various existing works show variable success, starting from detecting flaky tests to fixing them. Yet, despite these works, test flakiness continues to be a significant issue. While some solutions rely on rerun-based approaches that introduce overhead or depend on extra metrics like code coverage, others struggle to scale effectively across varied codebases. Most importantly, several of these methods often fall short to align with the practical needs of developers as they need to detect flaky tests without any overhead costs. Given the critical role of tests in software development and the presence of test flakiness problem, there is a need for comprehensive and efficient solutions to detect and deal with test flakiness.

3.2 Thesis Statement

The thesis statement is that machine learning and data science can address *better* the problem of test flakiness in terms of detecting, living with, and categorizing test flakiness. To investigate this, I evaluate current methodologies addressing test flakiness based on the discussed problem statement. By understanding the needs from these evaluations, I discuss and propose solutions based on machine learning and data science to overcome the needs. I evaluate the thesis by discussing each proposed solution with research questions, which are detailed in following subsections.

3.2.1 Detecting Flaky Tests

I started my research in detecting flaky tests by conducting a rerun experiment to analyze this approach. During the experiment, I investigate the frequency of detecting flaky tests within a specific number of test suite runs and analyze the possibility of reproducing previously detected flaky tests by other studies. Follow this experiment, I investigate using machine learning techniques to predict if a test is flaky or not learning from other flaky tests in the test suite. I propose FlakeFlagger, a machine learning classifier to make predictions for new tests by utilizing data of both flaky and non-flaky tests. In addition to the purpose of FlakeFlagger, it could be valuable as it allows developers to minimize the cost of re-running tests by focusing on tests that FlakeFlagger predicts as being more likely to become flaky. To address the challenge of detecting flaky tests using my experiment of the re-run and FlakeFlagger, I am answering the following questions:

RQ 4.3.1: How many flaky tests can be found by rerunning tests given different rerun budgets?

RQ 4.3.2: How hard is it to reproduce a flaky test failure?

RQ 4.3.3: How effective is FlakeFlagger at predicting flaky tests?

RQ 4.3.4: How helpful FlakeFlagger's features in distinguishing between flaky and non flaky tests?

The related findings of the detection of flaky tests are detailed in Section 4. Specifically, the first two research questions are discussed in Section 4.1. The responses to research questions 3 and

4 are found in Section 4.2. The answers to all these questions are primarily summarized from the paper “FlakeFlagger: Predicting Flakiness Without Rerunning Tests”[**?**].

3.2.2 Living with Test Flakiness

Even with the detection of flaky tests, they continue to exist in test suites. Developers often keep these tests for various reasons, such as understanding these tests impacts or they may be relied upon to detect true (non-flaky) failures. Hence, for a given failure from a known flaky test, how to determine if the failure is flaky or true failure. Recent studies show that developers can recognize a failure is flaky by examining the failure message and stacktraces as they could have encountered flaky failures with similar failure message and stacktraces[**?**]. A recent study refer to the process of identifying two failures with matching failure messages and stacktraces as *failure de-duplication*. Based on this approach, I am studying the failure logs as a source to compare a new failure with both flaky and true failures and using approaches based on *failure de-duplication* to determine if the failure is flaky or not [**?**, **?**]. As it is possible not to have previous flaky failure to compare with, I have proposed machine learning classifiers to learn from already existed flaky and true failure from other flaky tests. This lead me to formulate the following research questions:

RQ 5.3.1: How often are flaky failures repetitive?

RQ 5.3.2: With prior flaky and true failures, is it feasible to use the failure de-duplication to tell if a failure is flaky or true one?

RQ 5.3.3: How far utilizing machine learning being helpful in finding the differences between flaky and true failures?

The research and its findings are detailed in Section 5. Specifically, Section 5.1 introduces the proposed approaches for failure de-duplication. In Section 5.2, the methodology for addressing the research questions is discussed, and the findings are presented after answering the questions, as outlined in Section 5.3. This works is already submitted.

3.2.3 Categorizing Flaky Failures

Identifying the cause behind test flakiness can aid in assessing flaky failures effectively. In the remainder of my thesis, I am working on proposing a tool designed to categorize flaky failures based on failure logs by clustering failures where each cluster should represent one root cause. I will explore leveraging machine learning to learn from the features of collected failures and predict the root causes. This portion of the thesis is currently in the discussion phase, with particular focus on the process of data collection and the definition of the research methodology. This work remains flexible and may be adjusted to reflect the current trends and concerns within the broader research community, but I am initially focus to answer these research questions.

RQ 6.1: Is it possible for a flaky test to be triggered by multiple flakiness root causes?

RQ 6.1: Can failure logs associate flaky failures with their root causes using machine learning?

The discussion of these research questions, along with detailed insights on the topic of categorizing flaky failures, can be found in Section 6.

4 Detecting Flaky Tests

The detection of flaky tests is a crucial aspect of software testing and development. This process not only protect resources by preventing wasted efforts on resolving misleading failures but also influences software release timelines by providing clearer insights into test outcomes. When test failures are accurately identified as flaky, developers can make more efficient release decisions and avoiding unnecessary delays. My motivation to work in this field falls in the significance of flaky test detection in enhancing software development processes and quality. My goal is to analyze existing detection techniques, assess their strengths and limitations, and use these insights to propose more detection techniques.

Running tests many times is the traditional way to find flaky tests. Unfortunately, there is little industrial and academic guidelines regarding how many times to rerun each test in order to check if there are flaky or not. Prior studies consider many different numbers to run each test such as 10 [**?**], 16 [**?**], 100 [**?**] or, 4,000 times [**?**]. As there are various non-deterministic reasons behind flaky tests, it is hard to claim that developers will observe test flakiness within a fixed number of runs. Running tests could not be a major problem if developers can ensure that within e.g. 5 runs flaky tests can be detected. Another problem related to rerun is that it could be hard to reproduce the flaky failure detected in the original environment using another environment (e.g. developers who locally debug flaky tests which failed on a server). This problem of reproducing flaky tests is due the lack of knowledge about the non-deterministic source that cause flaky failures whether it is due to Java version, network speed, etc.

Alternatively, I am proposing FlakeFlagger, a Machine Learning (ML) approach to identify which tests in a test suite are flaky, *without* rerunning them many times. FlakeFlagger learns from existing flaky tests in order to predict unseen tests if they are flaky or not. FlakeFlagger can be used to search for flaky tests in a large test suite, where developers identify that a portion of the test suite is or is not flaky, and use FlakeFlagger to help label the rest of the tests as flaky or not. Also, FlakeFlagger could help in terms *prioritize* which tests should be run first by reporting which tests are most likely to be flaky.

By proactively identifying flaky tests, I may also help developers understand why these tests are flaky. Prior work has suggested different properties of tests that might make them more likely to be flaky, and FlakeFlagger can report which of these features are present in each test [**?**, **?**]. In practice, if a feature has a strong correlation with flakiness, developers might choose to focus on this feature in their future test maintenance and development activities.

4.1 RERUN: Empirical Study

To gather data on flaky tests, 24 Java projects were selected and run, some of which had been previously studied for test flakiness using different revisions [**?**] [**?**]. The entire test suites of these projects were run 10,000 times, which differed from the previous work [**?**] [**?**]. Only a single revision of each project was considered, which was either the most recent revision at the time of writing or the same revision studied in [**?**] or [**?**].

The rerun scripts were used to break down large experiments into smaller units called ”jobs,” which were executed on virtual machines. A single job was the execution of a Java test suite on a specific revision of a project using the Maven build system. For each job, the Maven build log and XML reports for each test run were saved. This approach aimed to create a level of isolation between test runs and simulate how an actual integration server would compile, test, and run a project's test suite.

The method I used to detect flaky tests through rerun is not the only approach available. There are other ways to increase the likelihood of detecting flaky tests. For instance, some flaky tests can be affected by the order in which they are run, and running them in different orders may uncover additional flaky tests [**?**]. However, this may also introduce a bias towards certain categories of flaky tests. Another way to detect more flaky tests is to run the experiment on different platforms and devices. However, my goal was to align the rerun experiment with standard development practices.

4.2 FlakeFlagger: Flaky Test Classifier

This section focuses on the construction process of the FlakeFlagger, beginning with the feature collection, followed by the classification process, and finally detailing how all of these elements have been designed as illustrated in Figure **??**.

4.2.1 Features Collections

Machine learning classifiers such as FlakeFlagger require a set of feature in order to learn and predict. I started with the prior work [**?**, **?**, **?**] to study which features is highly linked to flakiness. I aim to collect verity of features because of the fact that some flaky tests in different projects often have different root causes for their flakiness[**?**]. Similarly, some features that are predictive for one project may not be as predictive for others, due to the inherent non-determinism in flaky tests. I intentionally collect some dynamic features, in addition to static features (e.g. presence of textual tokens in the body of each test method). This is important because some causes not in the test method itself, but instead, in the production code that is executed by that test [**?**]. Ahmed et al. [**?**] categorized 23 developer-reported factors which affect test flakiness. These features are described by practitioners at a high level, and include test case complexity, hard-coded values and test smells. Eck et al. [**?**] interviewed 21 developers about flaky tests and tabulated the frequency of different kinds of flaky tests as well as developers' fixes for those flaky tests.

Inspired by previous studies on test flakiness, I developed a list of sixteen features, some of are based on general studies on the causes of flaky tests [**?**, **?**], while others are defined as bad practices in writing unit tests. Hence, I considered all of the features described in the prior works, and then selected only those for which I could write automated detectors. This ends up with implementing detectors for each of the features shown in Table 1. This list of features is not intended to be complete: there may yet be other features that can be easily collected and will be useful for predicting test flakiness.

While some of the features can be detected by inspecting the test method statically (specifically, the conditional logic smell and test line of code), the rest of the features require more than static analysis. A hybrid static/dynamic framework *detector* was developed to collect the statement coverage of each test, and then statically analyze the covered code in order to collect these behavioral features. The *detector* also collect a variety of other features related to the statement coverage of each test, such as how many recently changed lines of code are covered. The *detector* is implemented as an extension to the Maven build system.

Table 1: Complete list of features captured for test flakiness prediction. The Covered Lines Churn feature is represented in multiple forms based on the h values (number of the past commits). In the evaluation, I considered $h = 5, 10, 25, 50, 75, 100, 500$ and $10,000$

	Feature	Description
Test Smells	Indirect Testing	True if the test interacts with the object under test via an intermediary [?]
	Eager Testing	True if the test exercises more than one method of the tested object [?]
	Test Run War	True if the test allocates a file or resource which might be used by other tests [?]
	Conditional Logic	True if the test has a conditional if-statement within the test method body [?]
	Fire and Forget	True if the test launches background threads or tasks. [?]
	Mystery Guest	True if the test accesses external resources [?]
	Assertion Roulette	True if the test has multiple assertions [?]
	Resources Optimism	True if the test accesses external resources without checking their availability [?]
Numeric Features	Test Lines of Code	Number of lines of code in the test method body
	Number of Assertions	Number of assertions checked by the test
	Execution Time	Running time for the test execution
	Source Covered Lines	Number of lines covered by each test, counting only production code
	Covered Lines	Total number of lines of code covered by the test
	Source Covered Classes	Total number of production classes covered by each test
	External Libraries	Number of external libraries used by the test
	Covered Lines Churn	h-index capturing churn of covered lines in past 5, 10, 25, 50, 75, 100, 500, and 10,000 commits. Each value h indicates that at least h lines were modified at least h times in that period.

4.2.2 Classification Process

FlakeFlagger takes a list of tests, where each of them is represented as a vector $\{x_1, x_2, x_3, \ldots, x_n\}$ where each *x* represents a feature value and *n* correspond the total number of features. I applied data inspection and cleaning process to make that dataset more clear for the classification. Missing data can exist to the dataset due to the fact that some features collected by the detector can be incomplete e.g. due to crashes in the middle of the test execution. Some tests are not written in Java, and hence the feature detectors may not be applicable to them, and due to inheritance, some tests may not have source code in the project under test.

As in any classification problem, considering multiple supervised learning algorithms could be better. In FlakeFlagger, it is designed to use a set of models including Random Forest (*RF*) and Decision Tree (*DT*. I follow a feature selection process using *information gain*, which computes the amount of information that a feature can provide for a classification [?]. Imbalanced datasets (where there is not an equal number of instances in each class — flaky and non-flaky tests in my case) usually have very low information gain values.

4.2.3 Experimental Design

To evaluate FlakeFlagger, I used the same dataset described in Section 4.1. I ran the detector described in Section 4.2.1, to collect the set of features shown in Table 1. FlakeFlagger, similar to any machine learning classifiers, relies on two data sets: one to build the model (training) and another for testing. Because this is not already designed and I have only one dataset, I applied k-fold cross validation [?] to evaluate the model. Following this practice, I split the data into k parts, leave one part for testing and $k - 1$ to train the classifier. I then repeat this process with another $k - 1$ parts, each time leaving one part for testing. However, k-fold cross validation is most applicable to data that is evenly balanced, where the proportions of each class (flaky and not flaky) are similar. In fact, most tests are not flaky, which means imbalance data. To overcome this, I applied a sample technique SMOTE [?] only on training dataset, to ensure a valid and fair result.

In my prediction evaluation, I label each prediction result as a True Positive (TP), False Negative (FN), False Positive (FP), or True Negative (TN) as follows: TP - predicted flaky, known to be flaky; FP - predicted flaky, not known to be flaky; FN - predicted not flaky, known to be flaky; TN - predicted not flaky, not known to be flaky. I also evaluate the models using F1-score, which is computed using the standard formula based on Recall and Precision. Lastly, I calculate the Area Under the Curve (AUC), a measure of how effective a model is at distinguishing classes.

In the evaluation, false positives represent the number of tests that might be considered as flaky by developers, resulting in excess effort spent re-running them to determine if they are flaky or not. I focus primarily on total positives, because I have confidence that the collected flaky tests are indeed flaky, but I cannot be confident in my classification of a test as not flaky. In other words, the oracle is a result of detecting flaky tests after 10,000 runs for each test, but this does not guarantee that the "not flaky" tests are really not flaky: they may just not have been observed to be flaky. This approach also allows us to confirm FNs are truly flaky tests because they fail at least once during rerun tests. However, because of the inherent non-determinism in flaky tests, I cannot construct a reliable oracle to evaluate TNs and FPs, but report them as-is.

4.3 Evaluation

I evaluate my findings in detecting flaky tests by answering the following main research questions:

RQ 4.3.1: How many flaky tests can be found by rerunning tests given different rerun budgets?

RQ 4.3.2: How hard is it to reproduce a flaky test failure?

RQ 4.3.3: How effective is FlakeFlagger at predicting flaky tests?

RQ 4.3.4: How helpful is each feature in distinguishing between flaky and non flaky tests?

By running each test 10,000 times, 811 flaky tests were detected: about 3.6% of the total number of tests were flaky. The number of flaky tests in 24 projects are not equally distributed. For example, 10 projects with less than 10 flaky tests, 4 of which have only one, and one with none. On the other hand, there are 4 projects which have more than 100 flaky tests. *Spring-boot* has 163 flaky tests, 20% of the total observed flaky tests. Table **??** summarizes these results.

4.3.1 How many flaky tests can be found by rerunning tests given different rerun budgets?

I calculated the probability that each flaky test would have been detected with fewer reruns. It is important to consider that it is not possible to state the probability due to the fact that there could be uncontrolled unaware conditions that cause the failure. As a result, only roughly a quarter of all of the flaky tests that I found in 10,000 runs would have been found with 10 reruns, roughly half with 100 reruns and roughly two thirds with 1,000 reruns.

Table **??** categorizing each flaky test as failing either between 0 and 10 times, between 10 and 100 times, between 100 and 1,000 times, and finally over 1,000 times (out of the 10,000 runs). These sets are represented by colors as shown in Table **??**. The *red* bar, which refers to tests that flake less than or equal 10 times, takes the majority in 8 projects. In general, I found more than 33% of total flaky tests fail in less than or equal 10 times, 22% of flaky tests fail more than 10 and less than or equal 100 out of 10,000. This suggests that flaky tests datasets with limited runs still not ensure to detect *most* flaky tests. Furthermore, I acknowledge that even after 10,000 re-runs, it is still possible that all flaky tests in this dataset have been detected.

4.3.2 How hard is it to reproduce a flaky test failure?

In the previous **RQ**, the rerun experiment aims to study the difficulty of identifying flaky tests by re-running them on the same platform. However, this does not capture the difficulty when a developer rerun the tests on different environment e.g. local machine. to meet this, I compared the set of flaky tests identified from the 10,000 reruns with those detected by prior researchers on the same versions of the same projects, but in different environments. The columns *DeFlaker* and *iDFlakies* in Table **??** show the total number of flaky tests that that paper reported on that version of that project, along with the number of those tests that were also found to be flaky based on the reruns. A blank entry indicates that the revision of that project that I executed did not have any flaky tests reported by the prior work. The DeFlaker dataset contains flaky tests from many revisions of each project, but I only studied a single revision of each project, and hence, it is possible that DeFlaker had not identified any flaky tests in that revision. The iDFlakies dataset consists of both order dependent tests (which are detected by shuffling execution orders), and non-order dependent tests (which are flaky regardless of execution order). Since I purposefully did not shuffle the execution order of the tests (as described above), I include only the non-order dependent tests from iDFlakies for comparison.

Comparing to DeFlaker, I found 10 flaky tests out of the 20 tests identified as flaky by DeFlaker. Unfortunately, the DeFlaker authors did not retain the build logs from their test runs, so I are unable to diagnose why those tests appeared as flaky to DeFlaker but not to my reruns. Comparing to iDFlakies, I found 28 flaky tests out of the 70 non-order dependent tests that I reran. In the case of iDFlakies, the authors *did* retain the build logs that show how these tests failed, and I confirmed by hand that the tests that I missed in my rerun experiment truly were flaky, and could have been detected as flaky if I had rerun them more. These results are indicative of the true non-determinism of flaky tests and the difficulties that developers face reproducing them: even with 10,000 reruns, I could not detect all flaky tests.

4.3.3 How effective is FlakeFlagger at predicting flaky tests?

I used the results from rerunning tests (Section 4.1) as the oracle for FlakeFlagger classification process, and ran the feature detector once on each of the same tests in order to gather the data needed to build a model. I considered several different approaches to process the data, and measure classifier performance with a confusion matrix, precision, recall, F1-score and AUC. Even I applied different classification algorithms and balance techniques, I found that the best prefroamcne was random forest model built using the SMOTE technique for balancing the training data (and using unbalanced testing data). I compare the result of FlakeFlagger classifier with the one of the state-of-the-art flaky test classifier, a vocabulary-based approach proposed by Pinto et al. [**?**] which extracts tokens from each test using a simple bag-of-words model. I considered a hybrid model that adds the token features to FlakeFlagger's features. I consider only projects that have at least 10 flaky tests to ensure I have enough flaky tests for training as shown in Table 2.

Overall, FlakeFlagger and the vocabulary-based approach both detected a very similar number

Table 2: Prediction performance for FlakeFlagger, the vocabulary-based approach, and the hybrid combination of both. The hybrid approach builds a model with both FlakeFlagger's and the vocabulary-based approach's features. I show the number of True Positives, False Negatives, False Positives and True Negatives, Precision, Recall, and F1 scores per-project. The AUC value is calculated after each fold where the reported value is the overall averages of AUC values after all folds. Projects with zero F1 values have very low numbers of flaky tests (less than 3 per project), and illustrate known limitations of FlakeFlagger.

		Flaky by	FlakeFlagger							Vocabulary-Based Approach [?]							Combined Approach						
Project	**Tests**	**Reruns**	**TP**	**FN**	**FP**	**TN**	**Pr**	**R**	**F**	**TP**	**FN**	**FP**	**TN**	**Pr**	**R**	**F**	**TP**	**FN**	**FP**	**TN**	**Pr**	**R**	**F**
spring-boot	2,108	160	139	21	15	1,933	90%	87%	89%	134	26	703	1,245	16%	84%	27%	143	17	18	1,930	89%	89%	89%
hbase	431	145	129	16	32	254	80%	89%	84%	89	56	152	134	37%	61%	46%	130	15	33	253	80%	90%	84%
alluxio	187	116	116	0	0	71	100%	100%	100%	108	8	11	60	91%	93%	92%	116	0	0	71	100%	100%	100%
okhttp	810	100	52	48	159	551	25%	52%	33%	79	21	444	266	15%	79%	25%	46	54	104	606	31%	46%	37%
ambari	324	52	47	5	3	269	94%	90%	92%	36	16	121	151	23%	69%	34%	47	5	3	269	94%	90%	92%
hector	142	33	30	3	8	101	79%	91%	85%	13	20	23	86	36%	39%	38%	25	8	11	98	69%	76%	72%
activiti	2,043	32	10	22	43	1,968	19%	31%	24%	12	20	531	1,480	2%	38%	4%	7	25	34	1,977	17%	22%	19%
java-websocket	145	23	19	4	1	121	95%	83%	88%	23	0	74	48	24%	100%	38%	19	4	4	118	83%	83%	83%
wildfly	1,023	23	11	12	27	973	29%	48%	36%	20	3	554	446	3%	87%	7%	17	6	24	976	41%	74%	53%
httpcore	712	22	14	8	23	667	38%	64%	47%	16	6	375	315	4%	73%	8%	15	7	24	666	38%	68%	49%
logback	805	22	3	19	17	766	15%	14%	14%	10	12	259	524	4%	45%	7%	5	17	11	772	31%	23%	26%
incubator-dubbo	2,174	19	8	11	35	2,120	19%	42%	26%	11	8	813	1,342	1%	58%	3%	13	6	23	2,132	36%	68%	47%
http-request	163	18	12	6	6	139	67%	67%	67%	16	2	84	61	16%	89%	27%	12	6	6	139	67%	67%	67%
wro4j	1,135	16	4	12	2	1,117	67%	25%	36%	2	14	101	1,018	2%	12%	3%	0	16	1	1,118	0%	0%	0%
orbit	86	7	1	6	8	71	11%	14%	12%	6	1	32	47	16%	86%	27%	1	6	7	72	12%	14%	13%
undertow	183	7	2	5	8	168	20%	29%	24%	6	1	63	113	9%	86%	16%	3	4	8	168	27%	43%	33%
achilles	1,317	4	2	2	3	1,310	40%	50%	44%	0	4	0	1,313	0%	0%	0%	0	4	0	1,313	0%	0%	0%
elastic-job-lite	558	3	0	3	0	555	0%	0%	0%	0	3	34	521	0%	0%	0%	1	2	0	555	100%	33%	50%
zxing	345	2	0	2	2	341	0%	0%	0%	1	1	144	199	1%	50%	1%	0	2	2	341	0%	0%	0%
assertj-core	6,261	1	0	1	5	6,255	0%	0%	0%	0	1	6	6,254	0%	0%	0%	0	1	0	6,260	0%	0%	0%
commons-exec	55	1	0	1	1	53	0%	0%	0%	1	0	18	36	5%	100%	10%	0	1	1	53	0%	0%	0%
handlebars.java	420	1	0	1	5	414	0%	0%	0%	0	1	91	328	0%	0%	0%	0	1	0	419	0%	0%	0%
ninja	307	1	0	1	3	303	0%	0%	0%	0	1	50	256	0%	0%	0%	0	1	0	306	0%	0%	0%
Total	21,734	808	599	209	406	20,520	60%	74%	66%	583	225	4,683	16,243	11%	72%	19%	600	208	314	20,612	66%	74%	68%

of flaky tests (599 and 583 respectively, out of a total of 808 flaky tests), but the two approaches varied in terms of precision — FlakeFlagger had a far lower false positive rate with just 406, compared to 4,683 false positives from the vocabulary-based approach. Considering the initial use-case of a researcher or developer using FlakeFlagger to determine which tests to run time-intensive flaky test detectors on, using either FlakeFlagger or the vocabulary-based approach would result the same number of flaky tests eventually detected (that is, both have comparable recall). However, if a developer uses both models to detect tests that are most likely to be flaky (which are false positive tests), FlakeFlagger reports fewer rate than vocabulary-based approach (406 vs 4,683).

FlakeFlagger's performance varied across projects: some projects (e.g., alluxio), had perfect precision and recall, while on others (e.g., okhttp and activiti) the approach was less successful. I investigated more about the results per projects and the performance could vary due to many reasons. First, the training and testing dataset sizes vary from one project to another. Because each project has its own environmental assumptions, development patterns, and other unique characteristics, it is really difficult to create a single general-purpose approach for flakiness classifications. Another reason for why performance varies across projects may be that not all flaky tests have been labeled correctly — no rerun-based technique can guarantee to find all flaky tests (even after 10,000 reruns). The higher number of observed flaky tests in a single project does not guarantee that FlakeFlagger performs well. Some flaky failures are due to rare dependency conflicts and network failures that are not captured well from the features described in Table 1. For example, okhttp

Table 3: Information gain (IG) for FlakeFlagger and the vocabulary-based approach.

Vocabulary-Based Features		FlakeFlagger Features	
Feature/Token	IG	Feature	IG
Test Lines of Code	0.023	Execution Time	0.121
throws	0.022	Source Covered Lines	0.067
should	0.020	Source Covered Classes	0.057
exception	0.018	Covered Lines	0.034
mtfs	0.018	Covered Changes (past 75 commits)	0.029
runbuildfortask	0.017	Covered Changes (past 50 commits)	0.028
tfs	0.017	Covered Changes (past 100 commits)	0.028
run	0.016	Covered Changes (past 500 commits)	0.024
transitive	0.016	Test Lines of Code	0.023
ioexception	0.015	Covered Changes (past 10 commits)	0.018
tachyon	0.014	Covered Changes (past 1000 commits)	0.015
fileid	0.011	Covered Changes (past 5 commits)	0.011
if	0.011	External Libraries	0.011
actual	0.010	Covered Changes (past 25 commits)	0.010
someinfo	0.010	Fire and Forget	0.007
testutils	0.010	Number of Assertions	0.006
writetype	0.010	Resources Optimism	0.005
some	0.009	Mystery Guest	0.003
checkspring	0.009	Assertion Roulette	0.002
testfile	0.009	Conditional Logic	0.002
createbytefile	0.009	Indirect Testing	0.001
family	0.009	Test Run War	0.001
checkcommonslogging	0.009	Eager Testing	0.000

has a high number of false positives and false negatives. With a further inspection on this particular project, there is a group of tests had all failed in the same way due to the same dependency problem in one single run.

4.3.4 How helpful is each feature in distinguishing between flaky and non flaky tests?

I reported the the information gain of each feature in FlakeFlagger's model, and the top 23 features in the model built using the vocabulary-based approach to get more insight about the effectiveness of these features. As shown in Table 3, I noticed that features that considered dynamic behavior from each test (e.g., execution time, covered lines, and coverage of recently changed lines) had a far greater information gain than the tokens that were statically extracted from the test method bodies. I found that the top eight FlakeFlagger features each had a higher information gain than the highest gain vocabulary feature. In the model built using the vocabulary-based approach [**?**], the features with the highest information gain were: test lines of code, presence of the 'throws' Java keyword, and several tokens like 'should', 'exception', and 'mtfs', each with an information gain significantly lower than the top features in FlakeFlagger's model.

The majority of the flaky tests in the prior study with the 'job' token came from a single project, "oozie," which is *not* in my evaluation. At the same time, the majority of non-flaky tests with the token 'job' in the dataset were in the project "elastic-job-lite," which was not included in the prior evaluation. The co-occurrence of individual tokens with flaky tests can vary dramatically between projects. Terms that correlate with flakiness in one project can not be expected to correlate with flakiness in other projects — this is also evident from the limited number of projects which contain each token. Note that this finding only underscores the need for a large, balanced dataset of flaky tests: the DeFlaker dataset that Pinto et al. used contained *more* flaky tests than FlakeFlagger dataset (1,403 vs 810). However, a single project in that dataset ("oozie") contributed more than half of those flaky tests (856), which can make it extremely difficult to draw conclusions that can generalize beyond a single project, or beyond the dataset.

4.4 Summary

The result from rerunning tests emphasizes the importance of finding creative and automated tools to detect flaky tests that do not rely on rerunning them, since rerunning tests can be impractical in the necessary amount of time needed, and still may not observe all flaky tests. The experiment shows how it is hard to identify a fixed number of runs to observe flaky tests. I know that even running tests 10,000 times will *still* not guarantee that all flaky tests have been found, since I did not succeed in reproducing many flaky test failures observed in prior work.

The proposed machine learning classifier, FlakeFlagger, shows promising results in the field of flaky test detection. The aim of FlakeFlagger is not only to predict flaky tests but also to prioritize tests by considering tests that are most likely to be flaky first for further investigation. Additionally, the advantage of FlakeFlagger is its ability to be expanded to include additional features as demonstrated in the *Combined Approach* column in Table 8. Utilizing machine learning methods like FlakeFlagger could help identify flaky tests with fewer resources. Thus, FlakeFlagger presents a valuable adding in the area of flaky test detection.

5 Living with Test Flakiness

Flaky tests may exist in test suites even after being identified, as some may play a role in detecting true (non-flaky) failures. Hence, a significant concern is about distinguishing whether a particular failure is flaky or not. Despite the research on detecting flaky tests, the issue of identifying the flakiness of a failure itself remains unexplored. In light of this, I am currently investigating the feasibility of using failure logs for detecting flaky failures.

5.1 Matching Failures Logs

When tests fail, they produce output that can be useful for debugging the failure. Failure logs provide a detailed understanding of the origin of the failure. Hence, developers typically debug logs to better understand the failure cause. In detecting test flakiness, a recent survey shows that some developers may manually debug failures logs to tell if a failure is flaky or not [**?**]. Developers can recognize a failure is flaky by examining the failure message and stacktraces as they could have encountered flaky failures with similar failure message and stacktraces [**?**].

The failure message and stacktraces describe the cause of the failure and hence, it is reasonable to use them to judge the failure de-duplication. If two failures have the same failure messages and stacktraces, it intents to have the same cause. However, even if a failure log matches an existing flaky failure log, it is likely also important to determine if the failure log also matches a true failure. If the flaky failure should differ from the true failures, this raises a question: To what extent can a failure log be informative to find the differences between the two type of failures? I study if the proposed de-duplication based approaches could help developers to determine if a new failure is flaky or true failure

This failure output might include a specific failure message (that corresponds to a failed assertion), a stack trace, and/or other console output. In this article, I focus specifically on the output that is common to the test suites of all projects that I have studied: stack traces. It is worthwhile to consider the case of matching different failure logs from the same test, and also matching failure logs between different tests. On the one hand, there might be the greatest confidence in matching failure logs from the same tests. On the other hand, utilizing data from multiple tests can increase the chances that a matching failure is found. In this section, I propose two approaches designed to find a failure de-duplication. The first approach, named text-based matching, that use the text of failure logs to find the similarity of given two failures. The second approach is called the Failure Log Classifier which adopts machine learning to predict if a failure is flaky or true failure.

5.1.1 Text-Based Matching

The text-based matching is my application of classic failure de-duplication approaches [**?**, **?**], where I de-duplicate failures by matching common stacktraces. This approach is also motivated by grey-literature suggestions that, "sometimes it's obvious to engineers that a test is flaky just by looking at the exception type and message" [**?**]. Intuitively, if an engineer has repeatedly seen the same flaky failure symptoms, they may be able to guess that a new failure is also flaky. As a result, developers with different experience, can leverage this approach to compare failures. This also speeds up the comparison process and enhances the reliability of comparison result.

I implement text-based matching by creating a dataset of parsed failure logs for each test. Each failure log is represented by its failure message and stacktraces. In terms of a failure message, it consists *exception type* (for example, AssertionFailedError) and everything follows this is treated as the *exception message*. For the stacktraces part, it is a set of lines representing the calls before the exception occurs and during the parsing, I are considering the top lines pointing directly to the test name. These lines reflect the most recent operations preceding the exception and often provide more details about the root cause of the failure.

I implement a pipeline to parse each failure into an XML file, cataloging all failures linked to a specific test. As shown in Listing 1, each failure block in the XML corresponds to one failure, containing four key components: the test name (**T**), exception type (**E**), exception message (**M**), and stacktrace lines (**S**). Within the **S** tag, individual lines are listed under the **line** tags, considering their order in the original log. If the test name is missing from the stacktrace e.g. fail in setup method, I consider the last line from the test class. For example, in Listing 1, the last line is not starting with the test name (present in **T**) but starts with the test class name. To categorize these XML files per project, the **T** tag includes a *project* attribute, referring the project name where the test belongs. In this phase, I also filter out non-deterministic stack trace lines internal to the JVM (e.g. `GeneratedMethodAccessor$XYZ` lines).

```
<Failure>
    <T project="alluxio">tachyon.JournalTest.TableTest</T>
    <E>UnknownHostException</E>
    <M>ip-172-31-48-81: ip-172-31-48-81: Temporary failure in name resolution</M>
    <S><line>java.net.Inet6AddressImpl.lookupAllHostAddr(Native Method)</line>
    <line>java.net.InetAddress$2.lookupAllHostAddr(InetAddress.java:929)</line>
    <line>java.net.InetAddress.getAddressesFromNameService(InetAddress.java:1324)</line>
    <line>java.net.InetAddress.getLocalHost(InetAddress.java:1501)</line>
    <line>tachyon.LocalTachyonCluster.start(LocalTachyonCluster.java:104)</line>
    <line>tachyon.JournalTest.before(JournalTest.java:33</line></S>
</Failure>
    ...
<Failure>
```

Table 4: A list of features used to train the Failure Log Classifier

Feature Name	Type	Description
Exception Type	Str	The name of the exception e.g. UnknownHostException
Test name in Stacktrace	Boolean	*True* if one of Stacktrace lines starts with the test name else *False*
Test Class name in Stacktrace	Boolean	*True* if one of Stacktrace lines contains the test class name else *False*
Other Tests in Stacktrace	Boolean	*True* if one of Stacktrace lines starts with other tests names else *False*
JUnit in Stacktrace	Boolean	*True* if one of Stacktrace lines starts with any Junit Lines else *False*
CUT in Stacktrace	Boolean	*True* if one of Stacktrace lines contains any lines from Code Under Test else *False*

```
        <T project="alluxio">tachyon.JournalTest.TableTest</T>
        <E>UnknownHostException</E>
        <M>ip-172-31-58-81: ip-172-31-58-81: Temporary failure in name resolution</M>
        <S><line>java.net.Inet6AddressImpl.lookupAllHostAddr(Native Method)</line>
        <line>java.net.InetAddress$2.lookupAllHostAddr(InetAddress.java:929)</line>
        <line>java.net.InetAddress.getAddressesFromNameService(InetAddress.java:1324)</line>
        <line>java.net.InetAddress.getLocalHost(InetAddress.java:1501)</line>
        <line>tachyon.LocalTachyonCluster.start(LocalTachyonCluster.java:104)</line>
        <line>tachyon.JournalTest.before(JournalTest.java:33)</line></S>
    </Failure>
```

Listing 1: Two flaky failures reported in Alluxio project after parsing their failure logs

The text-based matching relies on the text of the failure message and stacktraces. Within the discussed example in Listing 1, I found that the failure message (**M**) could contain information such as timestamp and IP address that make that failure unique by its test. For example, in Listing 1, different details like an IP address within **M** can set two failures apart. Hence, the text-based matching does not rely on **M**, and consider only stacktraces (**S**) and exception type (**E**). Given the challenges in capturing all potential cases where the failure message (**M**) could be identical, I avoid modifying these unique message details and discard the **M** during the comparison.

The main use of the text-based matching method is find a failure de-duplication. When given flaky and true failures, the text-based matching should be able to tell if a new failure is a de-duplication of flaky failures, true failures, or both. As this approach is design to find failure de-duplication within the same test, it could be useful to applicable across different tests especially if the failure stacktraces does not cover the test body, similar to the example provided in Listing 1.

5.1.2 Failure Log Classifier

There are cases where a newly written test introduces flakiness, or when there is no prior failures for reference. Motivated by these scenarios, I have proposed the Failure Log Classifier, which is trained on both flaky and true failures from *all* tests in a test suite. Then the classifier would be able to predict if the new encounter failure is flaky or true failure. For training the Failure Log Classifier, I considered a series of features, which represent each failure log. These features, shown in Table 4, are designed for generality. For instance, I ask whether the stacktrace lines cover any line in the test suite rather than the test itself. I chose the features based on the text of the failure logs. However, in order to collect these features, I analyze further as some features require knowledge of all test names in the test suite, test names throughout the entire project, and all source code file names of the code under test to facilitate determining the feature values for each test failure. Although other studies for predicting flaky failures use dynamic details [**?**], mygoal is to determine if relying on the information in failure logs can effectively predict flaky failures.

The Failure Log Classifier employed a simple *Decision Tree* (**DT**) as the supervised learning algorithm [**?**]. Based on the binary features used to train the classifier, decision tree provides a clear way to handle non-linear relationships. In addition to the **DT**, I use the naive bayes as well.

In terms of the trained dataset, I consider using the cross validation to split the whole dataset (flaky and true failures) to training set and testing set, as there are no provided separate testing dataset. If the dataset ends unbalanced (The number of flaky failures is less than true failures, or vice versa, I consider applying SMOTE [**?**] to balance the trained data and utilized stratified cross-validation [**?**] to ensure that the testing-fold part has at least one flaky failure.

For evaluating the Failure Log Classifier, I will use standard classification evaluation metrics, including the confusion matrix, precision, recall, and F1-score. These metrics will be calculated on a per-project basis to ensure that the results are not influenced by the diverse nature of different projects, as they do not represent a single domain.

5.1.3 TF-IDF

Term Frequency-Inverse Document Frequency (TF-IDF) is a commonly used numerical statistic that reflects how important a word is to a document in corpus [**?**]. Hence, I also consider applying TF-IDF for determining whether a failure is flaky or not based on matching other failures. TF-IDF has two components: Term Frequency (*TF*) which represents the frequency of a term (word) in a document and if a term appears frequently in a document, its *TF* will be high. Second, Inverse Document Frequency (*IDF*) which measures the significance of the term in the entire corpus and if a term appears in many documents, its *IDF* value will be low, reflecting its lower importance. The TF-IDF value of a term in a document is the product of its *TF* and *IDF* values. Equation 1 and 2 show the computation of *TF* and *IDF*, respectively.

$$\text{TF } (t) = \frac{\text{Number of times term } t \text{ in a document}}{\text{Total number of terms in the document}} \tag{1}$$

$$\text{IDF } (t) = \log(\frac{\text{Total number of documents}}{\text{Number of documents where } t \text{ in it}}) \tag{2}$$

In the context of studying failure logs, I refer *document* to a *failure* and the *t* to the token I extract from each failure message and stacktraces. For each failure in the generated XML file used in the text-based matching, I tokenize each line of each stacktrace (including the exception type) by split the words using the *dot* as separator (and removing the symbols such parentheses). For example, the last line in Listing 1 will be converted to the following set of tokens *(tachyon, JournalTest, before, JournalTest, java, 33)*. As mygoal was to evaluate the overall potential for this approach, I did not consider more advanced tokenization approaches [**?**].

5.2 Evaluation Methodology

The core contribution in this work is a rigorous empirical evaluation of the three flaky failure detection approaches described in the prior section.

5.2.1 Datasets

In order to effectively evaluate the failure de-duplication, I need a dataset that contains a large number of both flaky and true failures for the same test. The“FlakeFlagger” dataset was built by executing the test suites of 26 open-source Java projects 10,000 times and recording their outputs, yielding a large dataset of flaky failures [**?**]. I choose the FlakeFlagger dataset, as it contains the complete failure logs for each flaky failure, as opposed to other flaky test datasets like DeFlaker's [?] or iDFlakies [?]. Whereas a dataset of flaky failures can be mined by repeatedly running the same versions of the same tests, a dataset of true failures can only be mined from buggy code. While datasets of true failures *do* exist [?, ?, ?, ?], these datasets are typically intentionally constructed from tests that are *not* flaky (to make studying the defects easier). However, I are not aware of any accessible datasets that provide both flaky and true failures logs for the same set of tests. Even if one were to mine failures of flaky tests, there would still be a tremendous dataset imbalance problem: there tend to be far more tests that only fail due to flakiness as opposed to those that might also reveal faults [?].

I propose a novel methodology for constructing a dataset for this experiment, based on mutation testing. Mutation testing runs a program's test suite on generated mutants (variants of the program under test), and evaluates how many of those mutants are detected by a failing test. Mutants have been shown to be an effective substitute for real faults in software testing [?]. Hence, for each of the flaky tests in the dataset, I use mutation testing to build a large dataset of failure logs for true failures. To avoid contaminating the true failure dataset with flaky failures (caused by tests failing due to flakiness on the mutated code), I apply Shi et al.'s approach for filtering flaky mutants [?].

Hence, the dataset for the experiment consists of all of the flaky failures extracted from the FlakeFlagger dataset [?], supplemented by true failures generated by executing Shi et al.'s version of the popular PIT mutation testing tool [?, ?]. This modified version of PIT is configured such that each test-mutant failure is confirmed by re-running the test on that mutant, 20 times. Each failure that is deterministically reproduced is included in the dataset of failures. This confirmation step is necessary to filter out any flaky failures from the mutation dataset, and is used only for confirming that the failure is deterministic (I do not include each failure 20 times from each of the confirmation runs). Then from the collected failure logs of each killed mutant, I collect the failure messages and stacktraces. I extend the XML file per test to include a list of killed mutants, each of them contains the failure message and stacktraces.

In practice, flaky failures tend to be far more common than true failures. Given that the failure message and stacktraces includes the name of each test, the performance of any failure classifier could be misrepresented by a dataset that contained a large proportion of tests that *only* failed due to flakiness. For example, in a 9-month period observing Google's Chromium CI, Haben et al. observed that 1,446 tests failed with only true ("fault-revealing") failures, 22,477 failed with only flaky failures, and 897 failed showing both failures. A predictor based on the historical flaky failure rate of a test would easily have quite high recall at predicting flakiness (e.g. having at most $897/22,477 = 4\%$ true failures incorrectly labeled as flaky). My goal is to evaluate the performance of approaches that rely primarily on the failure message and stacktraces, and *not* just the historical flake rate of a test.

Hence, I include in the evaluation *only* tests with at least one flaky and non-flaky failure, and report the number of true and flaky failures in the dataset for each project. I were not able to successfully apply the PIT mutation testing tool to all of the projects despite significant efforts (one author expended at least 2 hours per-project to attempt to get it to work) — and hence, I were unable to gather a resource of failures for all projects. As a result, it is important to note that I do *not* include all projects or tests from the FlakeFlagger dataset. Whereas the FlakeFlagger dataset includes 811 flaky tests from 24 projects, I analyze only those tests for which I could collect a dataset of true failures: 543 flaky tests from 22 projects.

5.2.2 Research Questions

Using this dataset of 543 tests with both flaky and true failures, I design an experiment to answer the following research questions:

RQ1: How often are flaky failures repetitive? I discuss how frequently a flaky failure matches *at least* one other flaky failure, examining other flaky failures of the same test or other tests within the same project. By doing this, I show the repetition of flaky failures and the efficacy of the failure de-duplication approach.

RQ2: With prior flaky and true failures, is it feasible to use the failure de-duplication to tell if a failure is flaky or true one? The main objective is to evaluate the effectiveness of using text-based matching as an approach to find the differences between flaky and true failures. This helps practitioners and researchers if they can rely on the approach in detecting flaky failures. Since projects differ in their domain, root causes of flakiness, and the total number of flaky tests, I evaluate the approach on a project-by-project basis.

RQ3: How far utilizing machine learning being helpful in finding the differences between flaky and true failures? I aim to demonstrate the efficacy of employing machine learning classifiers in predicting whether a failure is flaky or not based on specific features extracted from failure logs. I are looking if a classifier can leverage failures from other tests within the same project to enhance the learning process of the model to better predict failures, especially from the newly written tests.

5.3 Result

5.3.1 RQ1: How often are flaky failures repetitive?

To answer this question, I being with the XML files that summarise all of the failures of each test (described in Sction 5.1). As the failures in each file correspond to either flaky or true failure, I count the number of flaky failures per test. I compute how many different flaky failures by their failure messages and stacktraces using the text-based matching approach as well as computing how many of flaky failure is repetitive (by the failure de-duplication with flaky failures within the same test or across all tests in the same project) and how many is not.

Table 5 summarizes the findings by considering the two cases: matching the flaky failures within the same test (shown in column *Per Test*), and matching the flaky failures across all failures from all tests in the same project (shown in the column *Across Tests*). By considering *Alluxio* as an example from Table 5: in the first case, there are 114 flaky tests and those tests cumulatively have 16,858 flaky failures in total (16,847 of them are repetitive and 11 are not). The 16,847 failures that were an exact match for at least one other failure represent just 310 unique failures. In the second case, the number of failures that are *not* repetitive dropped to 5 (16,853 repetitive failures). When comparing a new failure to flaky failures from different tests, the text-based matching might produce mis-match results due to lines in the stacktrace pointing to the test. To mitigate this, I exclude such lines during this type of comparison, ensuring a more accurate match result.

While I found that each flaky test in *Alluxio* could have different flaky failures, on average, each flaky test only had just over two different failures, each of which recurred many times. In most of

Table 5: Repetitive Flaky Failures within and across tests per project.
Failures column shows the number of flaky failures and the different failures (Set). The columns (1:n) and [1] refer to flaky failures that are and are not repetitives, respectively. Per Test refers to matching the failures within the same test. Across Tests refers matching all flaky failures from all tests.

		Failures		**Per Test**		**Across Tests**	
Projects	**Tests**	**Flaky**	**Set**	**[1]**	**(1:n)**	**[1]**	**(1:n)**
Alluxio-alluxio	114	16,858	310	11	16,847	5	16,853
square-okhttp	100	28,264	121	40	28,224	17	28,247
apache-hbase	62	19,822	100	14	19,808	5	19,817
apache-ambari	51	4,063	54	0	4,063	0	4,063
hector-client-hector	33	6,529	33	0	6,529	0	6,529
activiti-activiti	31	1,378	32	13	1,365	6	1,372
tootallnate-java-websocket	22	2,095	43	2	2,093	0	2,095
apache-httpcore	22	354	22	9	345	2	352
qos-ch-logback	20	438	21	8	430	4	434
kevinsawicki-http-request	18	3,501	18	3	3,498	0	3,501
wildfly-wildfly	18	50	18	12	38	4	46
wro4j-wro4j	14	10,833	21	3	10,830	2	10,831
spring-projects-spring-boot	12	14	13	12	2	5	9
orbit-orbit	7	2,943	7	0	2,943	0	2,943
undertow-io-undertow	7	92	12	3	89	1	91
doanduyhai-Achilles	4	165	5	1	164	1	164
elasticjob-elastic-job-lite	3	7	4	3	4	0	7
assertj-core	1	974	1	0	974	0	974
ninja-ninja	1	476	1	0	476	0	476
handlebars.java	1	411	1	0	411	0	411
apache-commons-exec	1	33	1	0	33	0	33
zxing-zxing	1	322	1	0	322	0	322
Total	543	99,622	839	134	99,488	52	99,570

the projects I studied in Table 5, there are a reasonable amount of repetitive flaky failures (by both considering the ratio of the number of flaky failures in column (**[1]**) to the total number of flaky failures or even to the set of flaky failures) as some projects (6 out of 22) have all flaky failures are repetitive. Hence, I conclude that, overall, flaky failures are extremely repetitive. While it is inappropriate to assume that each flaky test can only fail with a single set of symptoms, the number of unique failures is dwarfed by the frequency with which those failures recur.

I also carefully examine when flaky failures are not repetitive, and occur only once in the dataset. Across all the studied projects, there are only 134 out of 99,622 flaky failures (also out of 839 sets of flaky failures) that have never matched other flaky failures within the same project. Out of 134 that failed once, I found 95 of them are actually lack of the history of flaky failures (from tests that only failed once). Out of 22 projects, there are only two projects where the number of repetitive flaky failure is just equal or less than the number of non-repetitive flaky failures (*Elastic-job-lite* and *Spring-boot*), and all these failures are from tests that only fails once.

While it is common for frequently failing flaky tests to exhibit repetitive flaky failures, this trend is not consistent across all projects. For example, within the project *Hbase*, there are 6 flaky tests that failed more than 100 times have at least one non-repetitive flaky failure.

Table 6: The Text-Based Matching between flaky and true failures
The *Total Tests and Failures* column provides the total flaky tests, the number of true failures across these tests, and the count of flaky failures. The *Set of Failures* column displays the different failures within both flaky and true failures. The *Confusion Matrix and Evaluation By Failures* column presents the matching results between flaky and true failures by showing the confusion matrix per failures and the evaluation metrics. The # of Tests in TP and FN shows how many different tests in each one. The cumulative number of tests in *TP* and *FN* might exceed the total given in *Test* because a test might have multiple flaky failures in different categories.

	Total Tests and Failures			**Set of Failures**		**Confusion Matrix and Evaluation By Failures**							**# of Tests in**	
Project	**Test**	**True**	**Flaky**	**True**	**Flaky**	**TP**	**FN**	**FP**	**TN**	**P**	**R**	**SP**	**TP**	**FN**
Alluxio	114	32,608	16,858	6,491	310	9,615	7,243	1,694	30,914	85%	57%	94%	114	102
Okhttp	100	34,266	28,264	18,609	121	16,517	11,747	114	34,152	99%	58%	99%	58	53
Hbase	62	11,324	19,822	811	100	18,496	1,326	1,198	10,126	93%	93%	89%	58	14
Ambari	51	11,049	4,063	4,563	54	4,003	60	5	11,044	99%	98%	99%	50	2
Hector	33	3,604	6,529	1,769	33	1,382	5,147	12	3,592	99%	21%	99%	32	1
Activiti	31	46,100	1,378	16,018	32	932	446	2,609	43,491	26%	67%	94%	1	30
Java-websocket	22	1,299	2,095	330	43	591	1,504	816	483	42%	28%	37%	19	22
Httpcore	22	8,333	354	663	22	0	354	2,117	6,216	0%	0%	74%	0	22
Logback	20	2,614	438	903	21	56	382	368	2,246	13%	12%	85%	3	17
Wildfly	18	4,364	50	1,497	18	38	12	0	4,364	100%	76%	100%	6	12
Http-request	18	387	3,501	229	18	981	2,520	40	347	96%	28%	89%	4	14
Wro4j	14	540	10,833	90	21	800	10,033	29	511	96%	7%	94%	9	11
Spring-boot	12	2,150	14	244	13	2	12	0	2,150	100%	14%	100%	1	12
Undertow	7	2,304	92	236	12	8	84	940	1,364	0%	8%	59%	2	6
Orbit	7	822	2,943	302	7	87	2,856	57	765	60%	2%	93%	2	5
Achilles	4	442	165	245	5	120	45	46	396	72%	72%	89%	1	3
Elastic-job-lite	3	111	7	68	4	4	3	0	111	100%	57%	100%	1	3
Cmmons-exec	1	59	33	13	1	0	33	2	57	0%	0%	96%	0	1
assertj-core	1	17	974	9	1	974	0	0	17	100%	100%	100%	1	0
Handlebars.java	1	147	411	61	1	0	411	16	131	0%	0%	89%	0	1
Zxing	1	76	322	37	1	322	0	0	76	100%	100%	100%	1	0
Ninja	1	209	476	6	1	0	476	90	119	0%	0%	56%	0	1
22 Projects Total	543	162,825	99,622	53,194	839	54,928	44,694	10,153	152,672				363	332

I investigated whether specific exception types were associated with these non-repetitive flaky failures. From the dataset I analyzed, among the top 10 most frequently occurring exceptions, two exceptions appeared more frequently in non-repetitive failures than repetitive flaky failures. Specifically, the *RuntimeException* was observed 14 times out of its 23 non-repetitive cases, while the *SocketException* was also observed 19 times out of a total of 31 non-repetitive cases. I found that every failures with the *SocketException* was linked within the *Okhttp* project.

I observed that certain test suite runs, especially those with a higher number of failed tests, tend to exhibit repetitive flaky failures across most or all the failed tests. For instance, within the *Ambari*, 48 out of 51 flaky tests consistently failed together and 47 of these tests displayed the same failure messages and stacktraces each time they failed, and none of their stacktrace lines contain the test names.

Summary. Flaky failures are often repetitive. This can serve as an indicator for developers: previous flaky failures can be a reference to check if a newly encountered failure is familiar. However, there are *few* cases where a failure is not similar with any previously observed flaky failures. In such situations, a deeper investigation is needed to detect its flakiness. A valuable step in this investigative process involves comparing the failure with flaky failures from other tests, especially when the failure's stacktrace lines do not reference the test itself.

5.3.2 RQ2: With prior flaky and true failures, is it feasible to use the failure de-duplicaiton to tell if a failure is flaky or true one?

I investigate if the text-based matching can be used to determine if a failure is flaky or not based on the failure de-duplication. As I consider both flaky and true failures, I use the basic of confusion matrix as follow:

TP: Flaky failures that match at least one flaky failure and do not match any of the true failures.

FN: Flaky failures that match at least one true failure *or* does not match with any of the flaky failures.

FP: True failures that match at least one flaky failure.

TN: True failures that do not match with any of flaky failure.

This evaluation methodology follows the running use-case, where newly observed test failures are either labeled as flaky (and ignored), or triaged to developers for further debugging and analysis. I then evaluate the result of matching using the *Precision* (**P**), *Recall* (**R**),and *Specificity* (**SP**) as follow:

$$\text{Precision } (\mathbf{P}) = \frac{\text{TP}}{\text{TP + FP}} \tag{3}$$

$$\text{Recall } (\mathbf{R}) = \frac{\text{TP}}{\text{TP + FN}} \tag{4}$$

$$\text{Specificity } (\mathbf{SP}) = \frac{\text{TN}}{\text{TN + FP}} \tag{5}$$

I report, per each project, the confusion matrix as well as the scores of **P**, **R** and **SP**. To highlight if one result could be biased by the number of flaky tests as the number of times each test fails may differ, I show the number of flaky tests that forming each part of the confusion matrix result. This result is shown in the column *Flaky VS True* in Table 6.

I choose these metrics to evaluate the result as well as to reflect the use-case when a developer encounter a failure and decide to compare it with the historical flaky and true failures. Given a model where developers ignore test failures that are labeled as flaky, a safer approach would have a higher precision, as precision reports the frequency with which an approach falsely determines a test to be flaky. Since I consider scenarios where developers may be most concerned that there are few false positives, I also report specificity, which evaluates the percentage of true failures correctly labeled. Lower recall scores indicate that an approach inadvertently labels more flaky failures as true failures — indicating that a developer might spend more time debugging them.

I summarize the finding of using the text-based matching as a flaky failure detection approach. Table 6 shows the confusion matrix of using the approach as described in Section 5.2. The performance of the approach varies across projects. For example, there are projects with at least 95% precision (10 out of 22) while some projects with 0% (5 out 10 projects).

In projects where the text-based matching approach struggles to differentiate between flaky and true failures, a common thread emerges: these failures are typically presented as *assertion*

Table 7: Top 10 Most Occurrence Exception in Flaky and True Failures
The *Exception Occurrence* column details the frequency of a specific exception, indicating in how many projects, tests, and failures this exception has been observed. The *Match Result (with Stacktraces)* column displays the match distributions, considering stacktraces and the related test count while the, *Match Result (without Stacktraces)* column indicates match results based on exception types, excluding stacktraces.

	Exception Occurrence					Match Result by Failures (with Stacktraces)				Match Result by Failures (without Stacktraces)			
Exception Name	**Projects**	**Tests**	**Failures**	**True**	**Flaky**	**TP**	**FN**	**FP**	**TN**	**TP**	**FN**	**FP**	**TN**
AssertionError	21	407	51,453	20,507	30,946	6,120	24,826	4,850	15,657	64	30,550	13,968	6,539
NullPointerException	22	498	49,906	41,709	8,197	1,644	6,553	449	41,260	34	8,163	7,913	33,796
IOException	7	257	20,097	15,963	4,134	3,614	520	519	15,444	28	3,141	3,717	12,246
RuntimeException	17	420	13,810	13,676	134	43	91	1,011	12,665	31	103	1,141	12,535
NoServerForRegionException	1	35	11,686	169	11,517	11,512	5	0	169	1,921	9,596	75	94
UnknownHostException	9	234	9,942	319	9,623	9,620	3	0	319	9,620	3	0	319
ActivitiException	1	30	9,893	9,821	72	0	72	614	9,207	0	72	3,094	6,727
IllegalArgumentException	17	401	9,052	9,049	3	0	3	190	8,859	0	3	212	8,837
AssertionFailedError	7	98	8,832	7,054	1,778	66	1,712	1,648	5,406	66	1,712	4,150	2,904
PersistenceException	2	30	8,581	8,580	1	0	1	164	8,416	0	1	398	8,182

exceptions. Example of these failures are *all* of the **FN** flaky failures in *Java-websocket*, *all* of the **FN** flaky failures in *Orbit*, and 98% of the **FN** in *Http-request*. Another type of exceptions is the *NullPointerException* as it is shown that about 90% of **FN** failures in *Alluxio*. Even with the availability of stacktraces in these failures, these exceptions remain challenging to be used in finding the differences between flaky and true failures. On the other side, the projects which have reasonable precision and recall scores (or at least precision scores) like the case in *Hbase*, there are a verity of different exceptions like *UnknownHostException* and *IOException*, and less likely to have general exceptions such as *assertion* and *NullPointerException*.

I conducted a qualitative analysis to see if certain factors influence the results of **RQ2**. I aim to validate the text-based matching's performance and establish the applicability to other datasets. This examination involves multiple factors that could affect the efficacy of text-based matching in distinguishing flaky failures from true ones. Key considerations include the proportion of true failures and the number of times that a test flakes. Thegoal is to show whether the approach's success in a specific project (over others) comes from its capability to differentiate failures or if external factors play a role.

I found that the ratio of true failures does not affect the performance of the approach such that the approach under-perform if there are high number of true failures as an example in the project *Http-request* when there is no failure labeled as TP. However, in *Wildfly*, there are 100% precision and SP scores even there are 4,364 true failures. Even the 12 FN are actually a single flake failures. I found also in*Http-request* project, there are 72% of the flaky failure were labeled as **FN** even there are few number of the true failures (3,501 flaky vs 387 true failures).

For projects which have more than one flaky tests, I found the project *Ambari* has a significant result in **precision**, **recall**, and **SP**. I found that the majority of the flaky failures with the exception *ProvisionException*. As discussed in **RQ1**, the majority of flaky tests in this project failed together. Considering this case could be differ from the true failures because the cause of flakiness sounds to affect many tests (as most of these tests match each other by their failure messages and stacktraces.

From the Table 6, I show **TP** and **FN** values by tests. There are two projects (namely *Activiti* and *Hector*) where the all the failures labeled as **TP** and **FN**, respectively, come from one test.

With this observation, it is hard to say that all failures from one test which used to have its failures belong to one group intent to have all new failure with in the same already known labeled. There for, dealing with each of the failures have to be not influnced by the test name. For example, all flaky tests in *Alluxio* has at least one flaky failure labeled as **TP** and 102 of these tests have also failures with **FN** failures.

To gain insight into the value of matching stack traces (in addition to exceptions), I conducted an analysis to evaluate if the text-based matching approach is able to find the differences between flaky and true failures when only the exception type is considered, *without* the stacktraces. To analyze this, I extracted the most frequently failure exceptions types and evaluated how many of these could be helpful alone to find differences in various projects.

Table 7 presents the top ten most frequently occurring exceptions observed in the analyzed flaky failures. In some cases, the exception by itself cannot determine the differences of matching compared the case when I consider the stacktraces such as the case of *NullPointerException*. However, One of the reported exception, namely *UnknownHostException*, is still be able by the exception alone to find the differences between the flaky and true failures. It is important to clarify that the presence of an exception like *UnknownHostException* does not necessarily indicate a direct association with flaky failures. I have identified a few of the non-flaky failures reported with this exception in the *okhttp* project. Interestingly, none of the flaky failures in that project have been reported with an *UnknownHostException*. That means some exceptions could be linked to flakiness, but is not likely possible to draw a main rule across all projects.

In the experiment, I discovered some failures where exceptions match both flaky and true failures, as indicated in Table 7. In the context of the experiment, the most frequently occurring exception is the *AssertionError* which roughly 20% of these failures appear in **TP**. However, when considering only the exception type and excluding stacktrace lines, the proportion drops to less than 2%. The reason behind this observation is the generality of the *AssertionError* exception. For example, a test may have multiple assertion statements, and if they fail for different reasons, they match the exception but differ in the stacktrace. Therefore, it becomes challenging to attribute this type of exception to a specific type of failure.

Summary. I found that using the de-duplication approach to find flaky and true failures effective in some projects especially when their failures logs more informative than just assertion failures. For most of the project, relying on the stacktraces in addition to the exception type is helpful as most failure exceptions could be seen in both flaky and true failures.

5.3.3 RQ3: How far utilizing machine learning being helpful in finding the differences between flaky and true failures?

I design the evaluation on the confusion matrix of the classifier as follow: True Positive (TP) when a flaky failure is correctly predicted as flaky, False Negative (FN) for flaky failures predicted as true failures, False Positive (FP) for true failures predicted as flaky, and True Negative (TN) when true failures are correctly identified. From this matrix, I then calculate the precision, recall, and F1-score.

I employ the Failure Log Classifier using two classifiers (decision tree and Naive bayes) and two ways of dealing with imbalance dataset. In terms of balancing the dataset, I use the SMOTE technique if the ratio of one type of failures is less than 10% of the total number of failures of the other type. I also consider using the dataset as it is without balancing. I use stratified cross-

Table 8: The Result of Failure Log Classifier and TF-IDF of Flaky and True Failures Prediction? The Failure Log Classifier and TF-IDF show (per project) the confusion matrix, precision (P), recall (R), and F1 score of the overall prediction result.

	Total Flaky Tests and Failures				**Failure Log Classifier**							**TF-IDF**						
Project	**Test**	**Failures**	**Flaky**	**True**	**TP**	**FN**	**FP**	**TN**	**P**	**R**	**F1**	**TP**	**FN**	**FP**	**TN**	**P**	**R**	**F1**
Alluxio-alluxio	114	49,466	16,858	32,608	16,014	844	1,104	31,504	93%	94%	94%	16,580	278	394	32,214	97%	98%	98%
square-okhttp	100	62,530	28,264	34,266	28,123	141	1,585	32,681	94%	99%	97%	28,238	26	108	34,158	99%	99%	99%
apache-hbase	62	31,146	19,822	11,324	19,782	40	369	10,955	98%	99%	98%	19,676	146	19	11,305	99%	99%	99%
apache-ambari	51	15,112	4,063	11,049	4,055	8	482	10,567	89%	99%	94%	4,063	0	5	11,044	99%	100%	99%
Hector	33	10,133	6,529	3,604	6,529	0	405	3,199	94%	100%	96%	6,529	0	13	3,591	99%	100%	99%
activiti-activiti	31	47,478	1,378	46,100	947	431	311	45,789	75%	68%	71%	1,013	365	60	46,040	94%	73%	82%
apache-httpcore	22	8,687	354	8,333	315	39	110	8,223	74%	88%	80%	314	40	16	8,317	95%	88%	91%
Java-websocket	22	3,394	2,095	1,299	2,082	13	721	578	74%	99%	85%	2,082	13	722	577	74%	99%	84%
qos-ch-logback	20	3,052	438	2,614	172	266	104	2,510	62%	39%	48%	239	199	41	2,573	85%	54%	66%
Http-request	18	3,888	3,501	387	3,498	3	124	263	96%	99%	98%	3,498	3	54	333	98%	99%	99%
wildfly-wildfly	18	3,895	48	3,847	0	48	0	3,847	0%	0%	0%	48	0	0	3,847	100%	100%	100%
wro4j-wro4j	14	11,373	10,833	540	10,833	0	65	475	99%	100%	99%	10,833	0	29	511	99%	100%	99%
Spring-boot	12	2,164	14	2,150	6	8	0	2,150	100%	42%	60%	10	4	1	2,149	90%	71%	80%
orbit-orbit	7	3,765	2,943	822	2,943	0	69	753	97%	100%	98%	2,943	0	59	763	98%	100%	99%
Undertow	7	2,396	92	2,304	3	89	0	2,304	100%	3%	6%	5	87	0	2,304	100%	5%	10%
Achilles	4	607	165	442	120	45	0	442	100%	72%	84%	148	17	26	416	85%	89%	87%
Commons-exec	1	92	33	59	0	33	0	59	0%	0%	0%	33	0	2	57	94%	100%	97%
zxing-zxing	1	398	322	76	322	0	0	76	100%	100%	100%	322	0	0	76	100%	100%	100%
handlebars.java	1	558	411	147	411	0	16	131	96%	100%	98%	411	0	16	131	96%	100%	98%
assertj-core	1	991	974	17	974	0	1	16	99%	100%	99%	974	0	0	17	100%	100%	100%
ninja-ninja	1	685	476	209	476	0	90	119	84%	100%	91%	476	0	90	119	84%	100%	91%
21 Projects Total	540	261,810	99,613	162,197	97,605	2,008	5,556	156,641				98,435	1,178	1,655	160,542			

validation and leave one fold for testing purposes. Due to the limitation of showing all result, I picked the best performance. I opted out projects that had fewer than 10 total flaky failures to ensure that I have at least one flaky failure in each testing fold. Also, I came across a few tests from which I could not extract features, resulting in missing values. Given their minimal occurrence (two failure in *Wildfly*), I exclude these failures.

To further understand the efficacy of machine learning in this context, I looked for a state-of-the-art classifier based on the failure logs. Existing methods to detect flaky failures, like the work of Lampel et al. [**?**], do not align with the dataset, which is based on the failure logs. Given this and the discussed features, I considered an alternative baseline approach. I utilized TF-IDF to contrast the classifier's predictions. Furthermore, I investigated whether TF-IDF could serve as an alternative method, especially since the features of the Failure Log Classifier are directly from the syntax of the failure logs without involving dynamic features.

Table 8 shows the result of using the Failure Log Classifier and the TF-IDF in predicting a failure if it is flaky or not. While I considered two different classification algorithms (Decision Tree and Naive Bayes), I find that decisions trees (without any dataset balancing) performed the best. The relative performance of the Failure Log Classifier and the TF-IDF varies as in some projects they have at least 90% F1 scores with zero **FN** failures while in few projects it is worse than being randomly guessing. The performance of the two classifiers close to each others (97,632 **TP** in the Failure Log Classifier VS 98,426 in the TF-IDF **TP**). Both classifiers have less False Positive rates (5,553 in the Failure Log Classifier and 1,657 in TF-IDF) than the rate of using the text-based matching (10,153).

The main explanation of having better performance in terms of the total number of **TP** compared to the text-based matching in most projects is the ability to learn from other flaky tests in the same test suite especially with projects where flaky failures share mostly the same exceptions as the case in *Alluxio* where the majority of the flaky failures exception with *NullPointerException*. This is because, in my implementation of text-based matching, I do *not* remove test-specific lines from the stack trace. Future work might extend my approaches to abstract these elements out of the stack trace, making matches between tests more likely [**?**].

As the TF-IDF approach is motivated to be used as comparable approach to the Failure Log Classifier, it is roughly better than the Failure Log Classifier (in term of the number of false positive rates) but both classifiers outperform the text-based matching result reported in Table 6. The main explanation of having less false positives in the TF-IDF is the ability to have more information (e.g. line numbers), as discussed early in Section 5.1.3. I found including the stacktrace lines numbers added more values as reflecting different stacktraces. On the other side, the generality of the features that the Failure Log Classifier could be a reason that, even with high performance in most projects, still not outperform the the TF-IDF.

There are three projects where the Failure Log Classifier is completely under-perform the TF-IDF (as in *Wildfly* and *Commons-exec*). In the *Wildfly*, I found all flaky failures with the the exception *RuntimeException* with very low repetitive rate (each test at most 7 times of failures) while the same exception mostly appear in all true failures. This project performs well in the TF-IDF and even in the text-based matching. The main observation in these failures is that the line numbers in the tests differ, which is not captured from the features I proposed to train the Failure Log Classifier. In the *Commons-exec*, I have a similar situation with another exception type named *AssertionFailedError*. In a project where the classifiers and the text-based matching have very low performance (even with the TF-IDF) like in *undertow-io-undertow*, I found the majority of flaky failures *and* the true failures are forming with the exception *AssertionError*.

The usability of different machine learning approaches varies based on the specific use case and objectives. If the main goal is to maximize the number of true positives (**TP**) without being overly concerned about the rate of false positives, the approach with **TP** is the better. In this scenario, the model is more focused on correctly identifying as many flaky failures as possible, even if it means accepting a higher number of false positives. One of the main advantages of the Failure Log Classifier is its flexibility in extending the learned features. The model can be easily augmented with additional static and dynamic features extracted from each failure. The proposed features shown in Table 4 are not final but serve as a starting point, particularly utilizing information available within the failure log. By leveraging these additional features, the failure log classifier can potentially enhance its performance identifying flaky failures.

Summary. I found that both the Failure Log Classifier and TF-IDF are able to predict flaky and true failures in most the projects. I found TF-IDF is slightly better in terms of the total number of false positives and negatives failures compare to the Failure Log Classifier result.

5.4 Summary

I find that flaky test failures can be extremely repetitive — when a test fails due to flakiness, it is likely to match other flaky failures from the same or other tests. I apply approaches based on failure de-duplication [**?**, **?**], text-based matching, and simple machine learning classifiers. I find that, for some tests, these approaches can be extremely effective (with no false negatives or false positives), yet for other tests, these approaches are entirely ineffective. By examining attributes of tests and failures, I provide insights for future research on generalized approaches for detecting flaky failures.

6 Categorizing Flaky Failures

Throughout the rest of my PhD program, I plan to focus on categorizing flaky failures. This will involve the development and suggestion of tools to assist in the categorization process.

Addressing flaky tests, including the process of fixing them, faces challenges during software development. When developers come across a confirmed flaky test, their primary objective is to determine the appropriate action. The initial step developers take is to understand the reason causing a test being flaky. Numerous works have proposed categorizing flaky tests based on their root cause or predicting their categories[**?**][**?**]. Flaky test failures can arise from various reasons, and they may or may not share the same root cause. Once a flaky failure is confirmed as a flaky, how likely to be caused with the same root cause of previous flaky failures. I am exploring if two flaky failures root causes could be distingushedable by their failure logs (espcially the failure messages and stacktraces), and use them as indicator to identify the root causes of a new encountered flaky failure. If failures logs of flaky failures of the same tests differ because there are different root causes, I plan to utilize machine learning to construct a classifier that predicts the root causes of flaky failures. This prediction will be based on analyzing failure logs and dynamic information, which includes factors such as execution time.

6.1 Current Progress

I am searching for a dataset that labels flaky tests based on their root causes. Given the potential difficulty of obtaining such a dataset, I have come across a dataset that relies on collected commits from GitHub [**?**]. I also have found another dataset that may categorize flakiness causes, focusing on whether they are order-dependent or not [**?**]. Due to the need to have failure logs for this purpose, I will start with two types of root causes (order dependent and non-order dependent) as reported in iDFlakies dataset [**?**]. I am exploring if the flaky tests could be caused by many root causes. I am exploring the accessibility of the failure logs of these flaky tests in order to find if a flaky tests which have been flaky for different root causes lead to have different failure logs. The initial research questions I am trying to investigate are as follow:

RQ1: **Is it possible for a flaky test to be triggered by multiple flakiness root causes?** Initially, I aim to validate whether a flaky test can exhibit flakiness due to a range of reasons. The first step in this validation process would involve examining if the same test has been reported in multi dataset with different root causes.

RQ2: **Can failure logs associate flaky failures with their root causes using machine learning?** I aim to determine if the root causes can be predicted using the features gathered from the failures, especially the information from the failure logs.

These research questions are currently in the discussion phase and might be revised.

7 Research Plan

During the Fall 2023, I will continue working on the proposed work and measure how likely the findings are motivated to investigate into new areas. In terms of writing my dissertation, I intend to defend it during 2024.

References