

Understanding the Lifecycle of Flaky Tests and Identifying Flaky Failures

Samaneh Malmir

A Thesis

In the Department of

Computer Science and Software Engineering

Presented in Partial Fulfillment of the Requirements

For the Degree of

Master of Computer Science (Computer Science)

at Concordia University

Montréal, Québec, Canada

May 2024

© Samaneh Malmir, 2024

CONCORDIA UNIVERSITY

School of Graduate Studies

This is to certify that the thesis prepared

By: **Samaneh Malmir**

Entitled: **Understanding the Lifecycle of Flaky Tests and Identifying Flaky Failures**

and submitted in partial fulfillment of the requirements for the degree of

Master of Computer Science (Computer Science)

complies with the regulations of this University and meets the accepted standards with respect to originality and quality.

Signed by the Final Examining Committee:

Dr. Diego Elias Chair

Dr. Diego Elias Examiner

Dr. se-Hsun Chen Examiner

Dr. Peter Rigby Supervisor

Approved by

Dr. Joey Paquet, Chair
Department of Computer Science and Software Engineering

_____ May 2024

Dr. Mourad Debbabi, Dean
Faculty of Engineering and Computer Science

Abstract

Understanding the Lifecycle of Flaky Tests and Identifying Flaky Failures

Samaneh Malmir

Software testing is a critical aspect of ensuring the quality of software. Ideally, tests should produce consistent results when being executed repeatedly on the same version of the software. However, certain tests may exhibit non-deterministic behavior, commonly known as “flaky tests”. These tests can provide ambiguous signals to developers and make the test results unreliable.

Despite being a recognized phenomenon for decades, academic attention towards test flakiness has only recently increased. The current dissertation aims to contribute to the advancement of research in two directions. First, we focus on predicting the lifetime of a flaky test, an issue that has been left unaddressed in the flaky tests research area. Secondly, we question the efficiency of previous studies in discerning flaky failures from legitimate failures, focusing on the Chromium build result as our dataset.

In our investigation of the historical patterns of flaky tests in Chrome, we identified that 40% of flaky tests remain unresolved, while 38% are typically addressed within the initial 15 days of introduction. Subsequently, we developed a predictive model focused on identifying tests with quicker resolutions. Our model demonstrated a precision of 73% and a Matthews Correlation Coefficient (MCC) approaching 0.39 in forecasting the lifespan class of flaky tests.

Furthermore, we discovered that current vocabulary-based flaky test detection approaches misclassify 78% of legitimate failures as flaky failures when applied to the Chromium dataset. The results also revealed that the source code of tests is not enough indicator for predicting flaky failures, and other execution-related features must be contributed for better performance.

Acknowledgments

I wish to take this opportunity to express my heartfelt appreciation to those individuals who have been integral to this remarkable journey.

First and foremost, I want to extend my sincere gratitude and admiration to my thesis supervisor, Dr. Peter Rigby. This work would have been extremely challenging without his unwavering guidance, support, and encouragement. His enduring patience and mentorship have been invaluable throughout every phase of this study, from conducting research to composing this thesis. I am sincerely grateful for his presence as a supervisor, and I couldn't have wished for a better mentor.

Additionally, I am grateful to Concordia University for providing me with the opportunity to engage in research activities, which have significantly enriched my academic journey.

Lastly, but certainly not least, I express my deepest appreciation to my family and friends for their love, support, and understanding through this challenge. Their encouragement has been a constant source of strength and motivation.

Contents

List of Figures	viii
List of Tables	ix
1 Introduction	1
1.1 Predicting the Lifetime of Flaky Tests	2
1.2 Discerning Flaky from Fault-triggering Test Failures	4
2 Literature Review	7
2.1 Detecting and root-causing flaky tests	7
2.2 Ranking Flaky Tests	9
2.3 Vocabulary Based Methods	10
3 Predicting Lifetime of Flaky Tests	13
3.1 Chromium	14
3.1.1 Overview	14
3.1.2 Example of a Flaky Test	15
3.1.3 Chrome Score System	16
3.2 Data	20
3.2.1 Determining Test Categories and Build Status	20
3.2.2 Allowable Test Outcomes	20
3.3 Metrics	23
3.4 Research Questions	23

3.5	Objectives and Methodologies	24
3.5.1	RQ1: Descriptive Statistics - What is the Frequency of Each Category of Flaky Test?	24
3.5.2	RQ2: Flakiness Survival: How has the number of tracked flaky tests changed over time?	25
3.5.3	RQ3: Predicting Test Life Time: How accurately can we predict the lifetime of a flaky test based on description	25
3.6	Experimental Results	27
3.6.1	RQ1: Descriptive Statistics	27
3.6.2	RQ2: Flakiness Survival: How has the number of tracked flaky tests changed over time?	29
3.6.3	RQ3: Predicting Test Life Time: How accurately can we predict the lifetime of a flaky test based on description?	32
3.7	Discussion	33
3.7.1	Model Optimization Process	33
3.7.2	Window Training	36
3.7.3	5-fold Cross Validation	37
3.8	Threats to Validity	37
3.8.1	Internal Validity.	37
3.8.2	External Validity.	38
3.8.3	Construct Validity.	38
3.9	Conclusion	38
4	Discerning Legitimate Failures from Flaky Failures	40
4.1	Chrome Process	42
4.2	Data	43
4.2.1	Dataset Features	46
4.3	Background	47
4.3.1	Bag of Words	47

4.3.2	Word2Vec	49
4.3.3	BERT Model	49
4.4	Original Study	51
4.5	Objectives and Methodologies	52
4.5.1	RQ1:How often does each test fail without identifying a fault?	52
4.5.2	RQ2: Can we propose a model to discern legitimate failures from flaky tests?	53
4.5.3	RQ3: Are some test suites easier to predict than others?	55
4.6	Experimental Results	57
4.6.1	RQ1:How often does each test fail without identifying a fault?	57
4.6.2	RQ2: Can we discern legitimate failures from flaky tests?	60
4.6.3	RQ3: Are some test suites easier to predict than others?	63
4.7	Threats to Validity	65
4.7.1	Internal Validity.	65
4.7.2	External Validity.	67
4.7.3	Construct Validity.	67
4.8	Conclusion	68
5	Conclusion	70
	Bibliography	73

List of Figures

Figure 3.1	An example of a flaky test caused by a timeout	17
Figure 3.2	Flaky Score Sample	18
Figure 3.3	An example of a build result on Chromium project	19
Figure 3.4	Decision tree representing how test outcomes are determined in a build by the Chromium CI.	21
Figure 3.5	Model Architecture	27
Figure 3.6	Frequency of different outcomes in test expectation file	28
Figure 3.7	Accumulated Number of Flaky Tests Over Time	30
Figure 3.8	Log Scale Survival Chart of Lifetime of Flaky Tests	31
Figure 4.1	Decision tree representing how test outcomes are determined in a build by the Chromium CI	42
Figure 4.2	Sample Build Result in Chromium	44
Figure 4.3	Architecture of model developed based on Bert	56
Figure 4.4	Cumulative flake rate for the 970 unique flaky tests	58
Figure 4.5	Distribution of flip rate values for versions	59
Figure 4.6	Distribution of flip rate values for tests	60
Figure 4.7	Confusion matrix for the model trained on source code and dynamic features	62
Figure 4.8	Box Plot of flake rate for different test suits	65
Figure 4.9	Box Plot of flake rate distribution for flaky failures vs. failed tests within each test suit.	66

List of Tables

Table 2.1	Overview of Studies on Flaky Test Prediction Based on Machine Learning Models	12
Table 3.1	Chromium Dataset Summary	28
Table 3.2	Result of the model, predicting tests fixed in ≤ 15 days (quick15) and ≤ 10 day (quick10)	33
Table 3.3	Result of using different configurations and features	34
Table 3.4	Result of the window training model, for window sizes of 6 months and 3 months	37
Table 4.1	Chromium Dataset Summary for one month (January 2022)	45
Table 4.2	Number of Tests for each category of Final Status	46
Table 4.3	Flakiness magnitude	46
Table 4.4	Description of dataset features	48
Table 4.5	Performance of Random Forest Model	61
Table 4.6	Model B Performance: Word2Vec with Weighted Logistic Regression	63
Table 4.7	Model C Performance: BERT	63
Table 4.8	Percentage of Test Suit Categories in Dataset	64

Chapter 1

Introduction

Continuous Integration (CI) is a widely adopted software engineering process that streamlines collaboration among developers by allowing them to frequently merge their changes into a shared code repository. By automating various aspects of the development life cycle, CI enables faster and more efficient software development. One crucial step in this process is regression testing, which aims to ensure that new changes do not introduce errors or break existing functionalities. Test suites are executed for every commit, and the test results serve as a signal for integrating the changes into the codebase or identifying potential issues. Flaky tests are tests that pass and fail on different executions of the same version of a code under test. However, as codebases grow and change frequency increases, the effectiveness of traditional continuous integration systems diminishes. This is especially true for large, fast-moving codebases like those at Google, which receive numerous code changes per minute and experience significant file modifications each day [1]. Google has tackled this issue by creating a unique continuous integration system that employs dependency analysis to identify and execute only the tests affected by a code change.

While software testing has long acknowledged flakiness as a prominent issue, research on this subject has only recently started to gain momentum over the past few years [2]. While most existing studies concentrate on predicting flaky tests, our focus lies in identifying fault-triggering flaky tests and also predicting the lifetime class of flaky tests, significantly aiding developers in the debugging process. Furthermore, a limitation of previous research on flaky tests is its reliance on small datasets containing only a few flaky tests, rendering the results less reliable in an industrial setting. In

our research, we use the Chromium dataset of flaky tests, consisting of thousands of instances, allowing for a thorough examination of their characteristics within a Continuous Integration (CI)-based environment.

In the following, we will introduce our research questions for each of our research topics.

1.1 Predicting the Lifetime of Flaky Tests

RQ1: Descriptive Statistics: What is the frequency of each category of flaky test?

The Chrome project runs thousands of tests every day and publishes the result, analysis, bug-fixing process, and flaky test data publicly, which makes it a great source for investigating flaky test mitigation strategies and practices in a large-scale test environment. The primary function of the web tests in Chrome is to serve as a regression test suite. In this context, the focus is placed not only on the accurate rendering of a page but also on ensuring that the page conforms to the defined expectations. Put differently, a greater emphasis is placed on detecting shifts in behavior rather than solely on assessing correctness. All web tests have “expected results”, or “baselines”, which define the expected result and may be one of several forms. This baseline file can be a text containing JavaScript log messages, WAV files of the audio output for Web Audio tests, or a screen capture of the rendered page as a PNG file. The baseline file lives alongside the origin test in the web test directory.

The expected outcomes for each test are stored in the `testExpectations` file and possible expectations are pass, fail, skip, crash, or timeout. Tests can have multiple outcomes, that is a combination of these 5 categories. Developers manually update this file based on the results of tests and available options for handling a failed test.

In order to prioritize tests for investigation, it is essential to understand the content of the `testExpectations` file, the distribution of the test categories, and the time it takes for a flaky test on Chrome to be resolved. Addressing this question is key to devising an effective framework for approaching the problem.

Results summary. By analyzing the content of `testExpectations` file, we discover that 40% of the flaky tests are never fixed. Of the flaky tests that are fixed or removed, 38% are in less

than 15 days.

RQ2: Flakiness Survival: How has the number of tracked flaky tests changed over time?

While existing studies have predominantly centered around investigating the root causes of flaky tests [3–5] and enhancing flakiness detection mechanisms [6–8], our research takes a distinctive approach. Rather than focusing on the origins or detection of flakiness, our objective is to establish a comprehensive understanding of the temporal aspect, specifically, the duration for which tests endure the flaky state.

To achieve this, we conduct an analysis spanning the entire history of changes made to the `testExpectations` file in 3 years. This examination allows us to understand the temporal dynamics of flaky tests by tracking their insertion, removal, or alteration in the `testExpectations` file.

Result summary. Our investigation yield noteworthy insights, indicating that except for two transient spikes, the overall count of flaky tests remains relatively constant over the extensive 3-year timeframe. Our historical analysis of flaky test occurrences provides valuable context for understanding the persistent nature of flakiness within the Chrome testing environment.

RQ3: Flaky lifetime prediction: How accurately can we predict the lifetime of a flaky test based on the issue description?

In this research question, our focus is on evaluating how effectively we can predict the duration of a flaky test’s persistence within the Chrome testing environment. The manual investigation and resolution of flaky tests by Chrome developers prompt our exploration into automatically suggesting tests that have a higher probability of quick resolution, based on historical data.

To achieve this, we employ a Random Forest model, to classify the lifetime of a flaky test into two categories: ‘quick to fix’ and ‘slow to fix/not worthy of investigation.’ The model’s outcome provides valuable insights into identifying tests that are likely to be swiftly resolved, allowing developers to prioritize their efforts efficiently.

Result summary. Upon applying our model to the test data, we observe a precision rate of 73% in predicting these specific categories of flaky tests. This promising outcome highlights the potential effectiveness of our approach in accurately predicting the lifespan of flaky tests based on

the description of the test added upon insertion.

1.2 Discerning Flaky from Fault-triggering Test Failures

RQ1: How often does each test fail without identifying a fault?

Developers create tests to identify faults in software code. However, encountering failing tests that don't contribute to fixing a bug can be challenging for developers and a resource drain for a company. This research question aims to provide insight into how common is this issue within Chrome.

To assess the extent of test flakiness in Chrome, we utilize two metrics: flake rate and flip rate, as introduced in a prior study [9]. Our analysis reveals that 50% of flaky tests have a flake rate below 0.09, indicating that half of them experience flakiness in less than 9% of their runs. Furthermore, in our dataset, 84% of flaky tests have a flip rate exceeding 0.8, suggesting that a significant majority of tests frequently alternate between two potential outcomes, indicating a high level of flakiness.

Results summary. In the study conducted by [9], flip rate was used as a metric for ranking flaky tests. However, when applying this measure to the Chromium context, we found that the values showed no significant variation. As a result, we concluded that this measure should not be relied upon for ranking flaky tests.

RQ2: Can we propose a model to distinguish legitimate failures from flaky failures?

In this research question, we aim to replicate and improve a recent and notable study focused on identifying fault-triggering flaky tests. Specifically, we refer to the work by Haben et al. [10] in which they introduce a model designed to distinguish fault-triggering test failures from flaky failures. We selected this specific study as our foundational reference for several compelling reasons. Firstly, it stands out due to its application within a large-scale industrial project, encompassing thousands of flaky tests, which significantly elevates the reliability of the study's findings. Furthermore, the research conducts a crucial investigation in the field of flaky test research, focusing on a significant aspect that has not received sufficient exploration in prior works. Rather than predicting flaky tests, Haben et al. [10] employ failure-focused prediction methods focusing on discerning flaky

failures from fault-triggering failures.

To begin, we replicate their study, where the authors claim their model can accurately distinguish legitimate failures from false alarms, achieving a Matthews correlation coefficient (MCC) of up to 95%. However, our investigation reveals a notable inefficiency in their proposed methodology. Specifically, the authors opt for a random approach to dataset splitting, a strategy that deviates from prevalent real-world practices. In our empirical analysis, when we adopt a time-sensitive splitting approach that aligns more closely with industry norms, the outcomes fail to yield promising results.

In our attempt to achieve satisfactory performance, we develop various models to address this issue. Despite the comprehensive nature of our approach and the wide range of models developed, the task of accurately predicting legitimate failures within the dynamic landscape of real-world software development settings proves to be exceedingly challenging.

Result summary. Our models indicate that vocabulary-based methods cannot distinguish between flaky failures and fault-revealing ones, and other features need to be incorporated into the model for better performance.

RQ3: Are some test suites easier to predict than others?

In this research question, our central focus is to test the performance of the previously established models on a subset of data. This involves a comprehensive examination of the inherent flakiness characteristics across various test suit categories, with a specific emphasis on exploring the potential to predict faults from instances of unreliable behavior within each category.

Additionally, our investigation extends to analyzing the frequency of unreliable failures among tests compared to fault-revealing failures within each category. The primary goal is to identify the best threshold for classifying tests into distinct categories of unreliable or legitimate failures, based on observed behavior patterns.

Result summary. We categorize tests into three groups based on their test suits: Layout Tests, Browser Tests, and Unit Tests. Layout tests make up 80% of the dataset, and the ratio of flaky failures to legitimate failures remains consistent across subcategories. Additionally, we observe that Unit tests exhibit the highest flake rate among all categories. In terms of prediction, due to the persisting challenge of an imbalanced dataset, the model struggles to accurately predict failures that

reveal faults within each category.

The following sections of this thesis are structured as follows: In Chapter 2, we present an extensive literature review on flaky tests. Chapter 3 delves into our investigation on predicting the lifespan of flaky tests. Our contributions to the challenge of distinguishing fault-revealing failures from flaky failures and the corresponding research questions are addressed in Chapter 4. Lastly, Chapter 5 outlines the research findings and discusses the encountered challenges throughout the study.

Chapter 2

Literature Review

Regression testing is a crucial step in the software development process that aims to validate modifications made to a software system while ensuring that existing functionality remains intact [4]. Despite the advancements in regression testing techniques, flaky tests remain a persistent challenge. Flaky tests, as studied by Bell et al. [11], exhibit non-deterministic behavior and introduce uncertainty into the regression testing process. The adoption of Continuous Integration (CI) practices has revolutionized regression testing by introducing automated and frequent integration of code changes, allowing developers to quickly detect and address potential regressions in software systems.

This section provides a comprehensive literature review on regression testing, flaky tests, and different approaches highlighting key concepts, challenges, and advancements in the field.

2.1 Detecting and root-causing flaky tests

Detection. At Facebook, Mateusz [12] proposes a predictive change-based test selection strategy to mitigate the overall infrastructure cost associated with testing code changes and it involves the selection of tests based on their dependencies on modified code. The results demonstrate that implementing this strategy leads to a significant reduction in the total infrastructure cost of testing code changes by a factor of two, while still ensuring that over 95% of individual test failures and more than 99.9% of faulty changes are accurately identified and reported to developers.

Lam et al. [6] introduce the iDFlakies framework, which identifies flaky tests through the re-arrangement and re-execution of tests, categorizing them as either order-dependent or non-order-dependent tests. The authors explore the effects of various test reordering strategies on the detection of flaky tests. After applying the framework to 683 projects and pinpointing 422 flaky tests, they found that executing tests under random orders detects the highest overall number of flaky tests.

Bell et al. [11] conducted a study where they employed code evolution and code coverage to ascertain if new test failures between two commits were triggered by flaky tests. The authors monitored the code coverage of recent alterations and automatically identified flaky tests that covered unchanged sections of the code but exhibited different outcomes compared to prior executions. Evaluating their algorithm across 96 Java projects, they effectively detected flaky tests with a minimal false alarm rate of 1.5

King et al. [8] aims to develop a static predictor for flaky tests. They propose a Bayesian network model that utilizes static test properties and past test results. Flakiness symptoms such as poor performance, poor maintainability, high failure rate, and high flip rate are considered features for prediction. By applying this model, they achieve a 65.7% accuracy in predicting flaky tests.

Root Causing. The research conducted by Lam et al. [3] focuses on identifying the root causes of flakiness. Their findings reveal that although the number of distinct flaky tests may be limited, the proportion of failed builds attributable to flaky tests can be significant. To alleviate the burden of flaky tests on developers, the authors present an end-to-end framework designed to identify flaky tests and comprehend their underlying causes. They develop a framework to collect logs from both passing and failing executions, allowing them to detect differences and pinpoint flaky tests along with their underlying causes. This framework employs a preliminary tool called RootFinder to detect differences in logs between passing and failing runs.

Another investigation conducted by Vahabzadeh et al. [13] delves into the origins of faulty tests by analyzing the JIRA bug repository. Their findings reveal that 21% of false alarms stem from flaky tests caused by issues such as Asynchronous Wait, Race Condition, or Concurrency Bugs. Similarly, Thorve et al. [14] conducted an empirical study focusing on Android application projects to uncover the underlying causes of flaky tests. They introduce three new factors contributing to flakiness—UI,

dependency, and program logic—in addition to previously identified causes like concurrency and network problems.

Furthermore, an empirical inquiry by Fabio et al [15] explored the correlation between flaky tests and test smells. Test smells such as Resource Optimism, Indirect Testing, and Test Run War were examined, revealing that 45% of the tests displayed some degree of flakiness, with 54% of these flaky tests being associated with code smells potentially contributing to the flakiness. Moreover, the study demonstrated the effectiveness of refactoring in resolving flaky tests.

2.2 Ranking Flaky Tests

While the majority of works focus on root causing and automated fixing [3,13,15], Kowalczyk et al [9] at Apple evaluate the level of flakiness. They developed two models to measure test flakiness. The first model assesses randomness using entropy, while the second model measures temporal variation by comparing pass-to-fail and fail-to-pass transitions with all possible transitions. They combined these metrics to calculate a test's flakiness score and applied the models to labeled data sets and Apple's service tests. they achieved a 26% reduction in flakiness with a 2.19% decrease in fault detection. However, the extent of improvement varied across different environments, reaching up to 82% flakiness reduction with a mere 0.002% loss in fault detection. The research is limited by its reliance on an artificially generated dataset with a deliberately high flakiness rate of 80%, which does not accurately reflect real-world scenarios where flakiness is typically much lower. In the case of the chromium project, the observed flakiness rate was less than 1%, highlighting the disparity between the artificial dataset used in this research and the actual flakiness rates found in industrial projects.

In the study conducted at Ericsson by Rehman and Rigby [16], the authors aim to quantify the occurrence of test failures that do not identify faults (NFF) to optimize the unit test pipeline. They calculated the NFF failure rate for each test and compared it to the stable NFF rate observed during the stabilization period of the prior release. Utilizing a binomial distribution, tests with a higher number of failures compared to the expected stable NFF rate were prioritized for re-run and further investigation. The authors found that by focusing on tests showing a significant increase from the

stable NFF rate, testers were able to reduce the number of tests investigated by 35% to 42%. This research applies the methodology to a real-world project with a substantial number of flaky tests, thus producing more reliable results that can be compared to a labeled dataset.

In recent years, there has been a significant amount of research exploring the application of machine-learning techniques for test prioritization purposes in software testing. For instance, Spieker et al. [17] propose a reinforcement learning approach that incorporates historical test information, such as test duration, previous execution records, and failure history, to prioritize test cases. Their findings indicate that after approximately 60 continuous integration (CI) cycles, their model discovers a prioritization strategy that performs comparably to basic deterministic methods. Another example of machine learning-based test prioritization is presented by Jahan et al. [18], who combine artificial neural networks with test complexity information and software modification details. Their proposed model exhibits improved fault detection rates when compared to existing prioritization approaches. Clustering methods have also been employed for test prioritization purposes. In the study described in [19], test cases are clustered based on their dynamic runtime behavior, aiming to reduce the number of pairwise comparisons. Applying this technique to different test suites resulted in enhanced fault detection capabilities compared to coverage-based techniques.

2.3 Vocabulary Based Methods

Pinto et al. [7] approach the problem of detecting flaky tests using information retrieval techniques. They extract the vocabulary of flaky tests and classify tests as flaky or non-flaky based on this vocabulary. Several machine learning techniques are employed, with Random Forest and Support Vector Machines demonstrating the best performance in predicting flaky tests based on source code features. The authors report a high F-measure of 0.95 in the prediction of flaky tests. In their study, Camara et al. [20] conduct a replication using diverse learning algorithms to evaluate the effectiveness of code identifiers in predicting test flakiness across various datasets, considering both intra- and inter-project contexts. The findings of their research highlight that the context of a project significantly influences the vocabulary of flaky tests. Consequently, the defined vocabulary lacks the

necessary level of generalization to reliably predict flaky tests in other contexts, potentially impeding the widespread adoption of code identifiers for predicting test flakiness. Flakify, proposed by Fatima et al. [21] is a black-box, language model-based predictor for flaky test cases. Flakify was developed by fine-tuning CodeBert, a pre-trained language model, using solely the source code of flaky tests. The research outcomes demonstrate that Flakify achieves promising F1 scores on both the FlakeFlagger and IDoFT datasets, resulting in significant reductions in the costs associated with debugging test cases and production code. Specifically, Flakify reduces the debugging cost by 25% and 64% for the test cases and production code of FlakeFlagger, respectively.

The work of Pinto et al. [7] was further replicated by Haben et al. [22] across three distinct dimensions. Firstly, they adopted a time-sensitive selection of training and test sets, aiming to better align the study with real-world usage scenarios. Secondly, they experimented with a dataset of flaky tests written in a different programming language, Python, instead of Java used in the original study. Thirdly, they explored the impact of different features extracted from the Code Under Test. The authors of the replication study found that the information present in the Code Under Test had a limited influence on the performance of the vocabulary-based models. Additionally, they observed that employing a more robust validation process consistently led to a decrease in performance compared to the reported results. Nevertheless, they concluded that vocabulary-based models can still be employed to predict test flakiness in other programming languages.

In the study by Aman et al. [23], a variety of Natural Language Processing (NLP) techniques were employed, including topic modeling and Doc2Vec to transform test cases into vector representations. These vectors were then utilized to measure the dissimilarity between pairs of test cases using diverse distance metrics (Manhattan distance, Euclidean distance, and angular distance). The test cases that exhibited the greatest dissimilarity were granted the highest priority. Subsequently, the remaining test cases were ranked based on their proximity to this prioritized set.

Overall, despite the potential of NLP techniques, the current use of NLP in test selection and prioritization context is very limited. [24]

Summary

Among the numerous methods for predicting flakiness, vocabulary-based approaches [7] - [22]

Table 2.1: Overview of Studies on Flaky Test Prediction Based on Machine Learning Models

Study	Model	Feature category	Features	Year
King et al. [8]	Bayesian network	Static & dynamic	Code metrics	2018
Pinto et al. [7]	Random forest	Static	Vocabulary	2020
Bertolino et al. [25]	KNN	Static	Vocabulary	2020
Haben et al. [22]	Random forest	Static	Vocabulary	2021
Camara et al. [20]	Random Forest	Static	Vocabulary	2021
Camara et al. [26]	Random forest	Static	Vocabulary	2021
Alshammari et al. [27]	Random forest	Static & dynamic	Code metrics & Smells	2021
Fatima et al. [21]	Neural Network	Static	CodeBERT	2021
Pontillo et al. [28]	Logistic regression	Static	Code metrics & Smells	2021
Qin et al. [29]	Neural Network	Static	Dependency graph	2022
Olewicki et al. [30]	XGBoost	Static	Vocabulary	2022
Ackli et al. [31]	Siamese Networks	Static	CodeBERT	2023

[32] [30] stand out as the most widely adopted [2]. These methods rely on machine learning models to forecast test flakiness by analyzing the occurrences of source code tokens within candidate tests. Intriguingly, prior research has identified these approaches as remarkably accurate, with the current state-of-the-art achieving accuracy levels surpassing 95% [7], [22], [20], [21], [26].

Simultaneously, vocabulary-based approaches possess characteristics of being both static and text-based. Consequently, they exhibit traits of portability, meaning they are tailored to a specific programming language, and interpretability, enabling users to discern the factors contributing to flakiness based on the keywords influencing the model’s decisions. This combination of precision, portability, and interpretability renders vocabulary-based approaches highly appealing, as they offer flexibility and ease of practical application.

Chapter 3

Predicting Lifetime of Flaky Tests

This chapter is an extended version of the paper that is accepted for publication in 2024 International Flaky Tests Workshop 2024 (FTW'24) [33], preserving the original content.

In Continuous Integration (CI) is a vital software engineering process that facilitates collaboration by allowing frequent integration of code changes into a shared repository. By automating various facets of the development life cycle, CI enhances the speed and efficiency of software development [34]. An important aspect of this process is regression testing, where test suites are systematically executed for each commit, ensuring that new changes do not introduce faults or disrupt existing functionalities. However, the scalability and efficacy of traditional CI systems tend to diminish as codebases expand and change frequency intensifies. This challenge is especially notable in large, dynamic codebases like those at Google, which witness a high volume of code changes per minute and substantial file modifications each month [4]. Google addresses this by employing a unique CI system that uses dependency analysis to selectively execute tests affected by code changes.

Estimating the time required to fix flaky tests represents a pivotal aspect of software development and quality assurance. Flaky tests not only disrupt the testing process but can also introduce significant delays in software release cycles, leading to increased costs and potential missed market opportunities. The ability to predict the duration needed for fixing flaky tests can empower development teams to allocate resources efficiently, prioritize bug fixes, and streamline the testing pipeline.

Remarkably, despite the critical role that this estimation plays in software development, there is a noticeable gap in the existing research literature. While numerous studies have delved into flaky tests' identification and root cause analysis, the specific task of estimating the time taken to resolve these issues remains unexplored. Bridging this gap is not only essential for enhancing testing efficiency but also for optimizing resource allocation and project planning in software development.

In this paper, we study the Chrome project that runs thousands of tests every day and publishes the result, analysis, bug-fixing process, and flaky test data publicly. The scale of this public data makes it an excellent source for investigating flaky test mitigation strategies and practices in a large-scale test environment.

In contrast to conventional testing approaches, Chrome employs a system with multiple “expected outcomes” for each test, documented in the `testExpectations` file. These anticipated outcomes include pass, fail, skip, crash, and timeout. Developers manually adjust the test expectation file as they modify their expectations for a specific test. Our dataset for this research comprises 78,450 tests with manually assigned expectations, among which 8,279 are labeled as “flaky.” The dataset is derived from the historical changes made to this file. It's noteworthy that tests may have multiple outcomes, such as failure or crash, and developers update the file based on test results and their examination of failures.

In this chapter, we delve into Chromium's approach to addressing flaky tests. We start by presenting our dataset and conducting an in-depth analysis to gain a more profound understanding of the data. Subsequently, we introduce our model, showcasing its performance and outcomes. Furthermore, we engage in a thorough discussion regarding the performance of various models, taking into account different features.

3.1 Chromium

3.1.1 Overview

Starting its journey in 2008, the Chromium web browser has grown into a significant open-source project, with a vibrant community of over 2,000 contributors and a codebase spanning 25 million lines. Google is at the forefront of this effort, although collaboration with various companies

and individual contributors fuels its progress. For its Continuous Integration (CI), the Chromium project relies on the LuCI platform, employing more than 900 parallelized builders tailored to different variations of Chromium with diverse settings and operating system targets.

Builders manage builds initiated by project commits, sometimes encountering queues due to their limited capacity. This dynamic approach accommodates the project's swift development pace, allowing multiple changes to be integrated into a single execution cycle. Each build encompasses detailed information such as properties, start and end times, and overall status, all meticulously recorded for future reference.

In its initial stages, the project employed a linear strategy for building and testing, resulting in prolonged testing periods and constraints in cross-platform testing of Chromium. To address these issues, a swarming infrastructure was introduced, involving 14,000 concurrent build bots. This architecture minimizes delays, ensuring tests are executed promptly and efficiently despite the substantial daily commit load. Currently, 47 testers manage Chromium test suites across distinct operating system versions, collectively comprising around 200,000 tests. The most extensive suites are `blink_web_tests`, which evaluate rendering engines, and `base_unittest`, housing over 60,000 tests each. Test outcomes, whether passes, reruns due to failure, or intermittent flaky behavior, are systematically documented. In the traditional approach, flaky failures, like other forms of failures, are typically considered issues that need to be resolved before the code can proceed to the next phase. However, in the case of Google Chrome, flaky failures do not halt the progression of the code to the subsequent stages in the Continuous Integration (CI) pipeline.

3.1.2 Example of a Flaky Test

Figure 3.1 shows a flaky test found in build 119,0392 of the Linux Tester. This test, `printing/webgl-oversized-printing.html`, ensures that no crash happens on the main thread of the rendering process when using the system. During its initial run, the test failed after running for 31 seconds, and the run status indicated a `TIMEOUT` error occurred. Upon its second execution, the test completed in 15 seconds but was labeled as unreliable due to its flakiness. Subsequently, a problem was reported in Chromium's bug tracking system regarding this issue. Developers clarified that the test, at times, triggers significant memory allocation in the GPU process, leading to intermittent Out-Of-Memory

(OOM) errors and GPU process crashes.

The TIMEOUT error, by its nature, raises concerns about potential test unreliability, as one might reasonably expect other executions of the same test to finish within the specified time limit. Moreover, we can also search for signs of test instability within the source code. Like many UI tests in Chromium, this test is managed by the testRunner, which is responsible for its automated execution. In line 19, we observe the testRunner invoking the waitUntilDone() function. In Chromium's web tests, terminology related to waiting is commonplace. Such keywords, for instance, could potentially be utilized by tools designed to detect test or failure flakiness.

3.1.3 Chrome Score System

Flaky tests waste developer resources and reduce the confidence developers have in tests. They are often difficult to fix because of their non-deterministic behavior. All tests that fail are re-run multiple times on Chrome, and if any of the re-runs is a pass, then the build can be integrated and the test is labeled as flaky. Flaky tests are a substantial problem on Chrome, and developers prioritized them for investigation. Equation 1 shows how the flaky score is calculated.

$$Score = Sum(flaketyweight * impactedCLs) + Sum(flaketyweight * totalflakes) \quad (1)$$

In this equation, the impacted CLs are the number of builds that have experienced this flaky test, the total flakes are the number of flaky test failures in the last week, and the weighted flake type is defined by the severity/negative impact that a flake has on the CI system. Chromium has defined different types of flakiness and assigns a weight to each type based on the severity and importance of the failure type. In the following we briefly mention each type.

The “false rejection” flaky type has the greatest weight of 100 because it is an unexpected failure that stops the build from being integrated. This type of flaky test can fail multiple times in a row without revealing a fault. Figure 3.2 shows an example of the flaky score for a sample test. We see that the sample test was a “false rejection” flake in 7 builds in the past 7 days and affected 7 Change Lists (CLs).

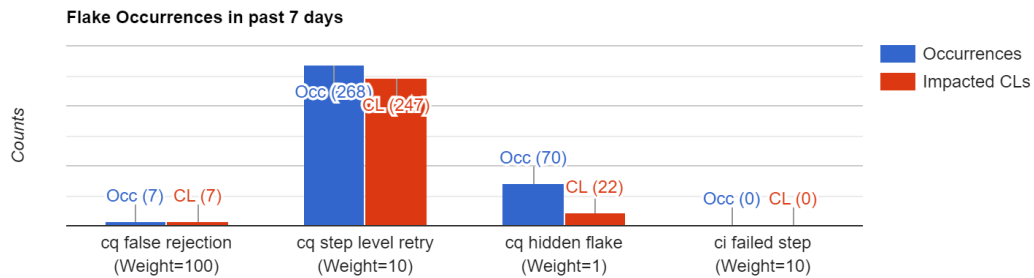
“CQ step level retry” weighs 10 and is an unexpected failure that causes additional retry steps.

```

1  <!-- This is a regression test for crbug.com/537711, in which printing
2      was causing a crash on the main thread of the render process -->
3  <head>
4  <script src="../../resources/js-test.js"></script>
5  </head>
6  <body>
7  <div id="console"></div>
8  <script>
9  var can = document.createElement('canvas');
10 can.width = can.height = 7326; // crbug.com/angleproject/6922#c15
11 document.body.appendChild(can);
12 var ctx = can.getContext("webgl");
13 ctx.clearColor(0, 0, 0, 1);
14 ctx.clear(ctx.COLOR_BUFFER_BIT);
15
16 if (!window.testRunner) {
17     testFailed("Requires window.testRunner");
18 } else {
19     testRunner.waitUntilDone();
20     testRunner.capturePrintingPixelsThen(() => {
21         testPassed("Printed without crashing.");
22         testRunner.notifyDone();
23     });
24 }
25 </script>
26 </body>

```

Figure 3.1: An example of a flaky test caused by a timeout. The test consists of an HTML file `printing/webgl-oversized-printing.html`, build 119,039 of the Linux Tester. The call to `waitUntilDone()` on line 19 is likely the reason for the failure.



$$3192 = 7 * 100 + 247 * 10 + 22 * 1 + 0 * 10$$

CLOSE

Figure 3.2: Flaky Score Sample

This type of test is categorized as Exonerated and will not block the build. We see in Figure 3.2 that in the past 7 days, the sample test flaked as a “CQ step level retry” 268 times impacting 247 CLs.

“CQ hidden flake” has the lowest weight and is a test that failed and re-run with one of the re-runs being a pass. This goal is to filter out noisy results. In Figure 3.2 the sample test was a “CQ hidden flake” occurred 70 times and it affected 22 CLs. As the test flakes across more CLs, its score increases, and it will eventually be investigated.

“CI failed step flake” is a rare type of test that involves the setup of the test failing. In Figure 3.2 the test has never flaked as “CI failed step flake”.


The tests with the largest impact and score are investigated, and a bug report is assigned to them [35].

Flaky failures can significantly impact build speed by requiring re-runs, which consume additional resources and may slow down the continuous integration process. Chrome experiences a high volume of new flaky test failures daily, often exceeding the available resources for resolution. For instance, as depicted in Figure 3.3, a single build may encounter as many as 212 flaky tests, a common occurrence. Consequently, developers face the challenge of prioritizing fixes to minimize resource consumption.

To address this issue, Chrome developers employ the testExpectations [36] file to temporarily suppress flaky tests during the investigation, helping to alleviate the burden on the continuous

Build [chromium](#) / try / [mac10.13-blink-rel](#) / 14948 | [Switch to the legacy build page](#)

Overview **Test Results** 1 Steps & Logs Related Builds Timeline Blamelist

 [Configure Table](#)

S	Name
▼	1 test variant: status= <i>UNEXPECTED</i> , test_suite= <i>blink_web_tests</i>
▼	<div style="display: flex; align-items: flex-start;"> <div style="margin-right: 10px;"> ! </div> <div> virtual/fenced-frame-mparch/wpt_internal/fenced_frame/pointer-lock.https.html history source ID: ninja://:blink_web_tests/virtual/fenced-fra... builder: mac10.13-blink-rel, test_sui </div> </div> <ul style="list-style-type: none"> > 3.4s run #1 unexpectedly failed in task: 5793d68cc215be11 > 3.2s run #2 unexpectedly failed in task: 5793d68cc215be11 > 2.8s run #3 unexpectedly failed in task: 5793d68cc215be11 > 2.3s run #4 unexpectedly failed in task: 5793d68cc215be11
▼	212 test variants: status= <i>FLAKY</i> , test_suite= <i>blink_web_tests</i>
>	! css3/filters/backdrop-filter-boundary.html
▼	<div style="display: flex; align-items: flex-start;"> <div style="margin-right: 10px;"> ! </div> <div> editing/pasteboard/data-transfer-items-image-png.html history source ID: ninja://:blink_web_tests/editing/pasteboa... builder: mac10.13-blink-rel, test_sui </div> </div> <ul style="list-style-type: none"> > 423ms run #1 unexpectedly failed in task: 5793d689fd3d8711 > 2.7s run #2 expectedly passed in task: 5793d689fd3d8711
>	! wpt_internal/css/selectors/focus-visible-select-001.html
▼	233 test variants: status= <i>EXONERATED</i> , test_suite= <i>blink_web_tests</i>
▼	<div style="display: flex; align-items: flex-start;"> <div style="margin-right: 10px;"> - </div> <div> compositing/gestures/gesture-tapHighlight-2-iframe-scrolled-inner.html history source ID: ninja://:blink_web_tests/compositing/ges... builder: mac10.13-blink-rel, test_sui </div> </div> <div style="background-color: #f0f0f0; padding: 5px; margin: 5px 0;">Unexpected passes are exonerated</div> <ul style="list-style-type: none"> ▼ 341ms run #1 unexpectedly passed in task: 5793d698052a4111 <ul style="list-style-type: none"> > Tags: monorail_component: Blink>Compositing, raw_typ_expectation: Failure, step_name: bli
>	- css3/blending/background-blend-mode-single-accelerated-element.html
>	- wpt_internal/bluetooth/requestDevice/filter-does-not-match.https.html
▼	0 test variants: status= <i>EXPECTED</i> , note: custom grouping doesn't apply to expected tests

Figure 3.3: An example of a build result on Chromium project

integration system.

In the subsequent section, we delve deeper into this approach, examining various potential outcomes and their frequencies.

3.2 Data

In the following sections, we describe our dataset, providing a comprehensive overview of the primary data source used in the study, and offering critical insights into the information underpinning the research findings.

We note that in prior work, the number of flaky tests per project was small. For example, Moritz et al. [37], studied Mozilla, which has 100 to 150 new flaky tests every week. In comparison, there are approximately 700 new flaky tests every day. This difference can be the result of the extensive codebase and a large number of contributors to the Chromium project. Also, Chrome’s rapid release cycle and continuous integration practices might introduce changes more frequently, increasing the chances of test flakiness.

3.2.1 Determining Test Categories and Build Status

Figure 3.4 outlines the decision-making procedure employed by LuCI (Chromium CI platform) to determine the result of a specific test within a build. A test is considered successful if it passes without errors in a single execution. In case of failure, LuCI initiates automatic retesting, repeating the test up to five times. If all reruns fail, the test is categorized as “unexpected”, leading to a build failure. Furthermore, if a test succeeds despite having experienced one or more failed executions within the same build, it is classified as “flaky.” However, this outcome does not hinder the overall successful progression of the build [38].

3.2.2 Allowable Test Outcomes

The scale of flaky tests on Chrome forces developers to find new ways to reduce the impact of flakiness on the CI environment. The *testExpectations* file is the solution. It records the expected, i.e. allowed, verdicts for each test. The *testExpectations* is a way to suppress the effect of flaky

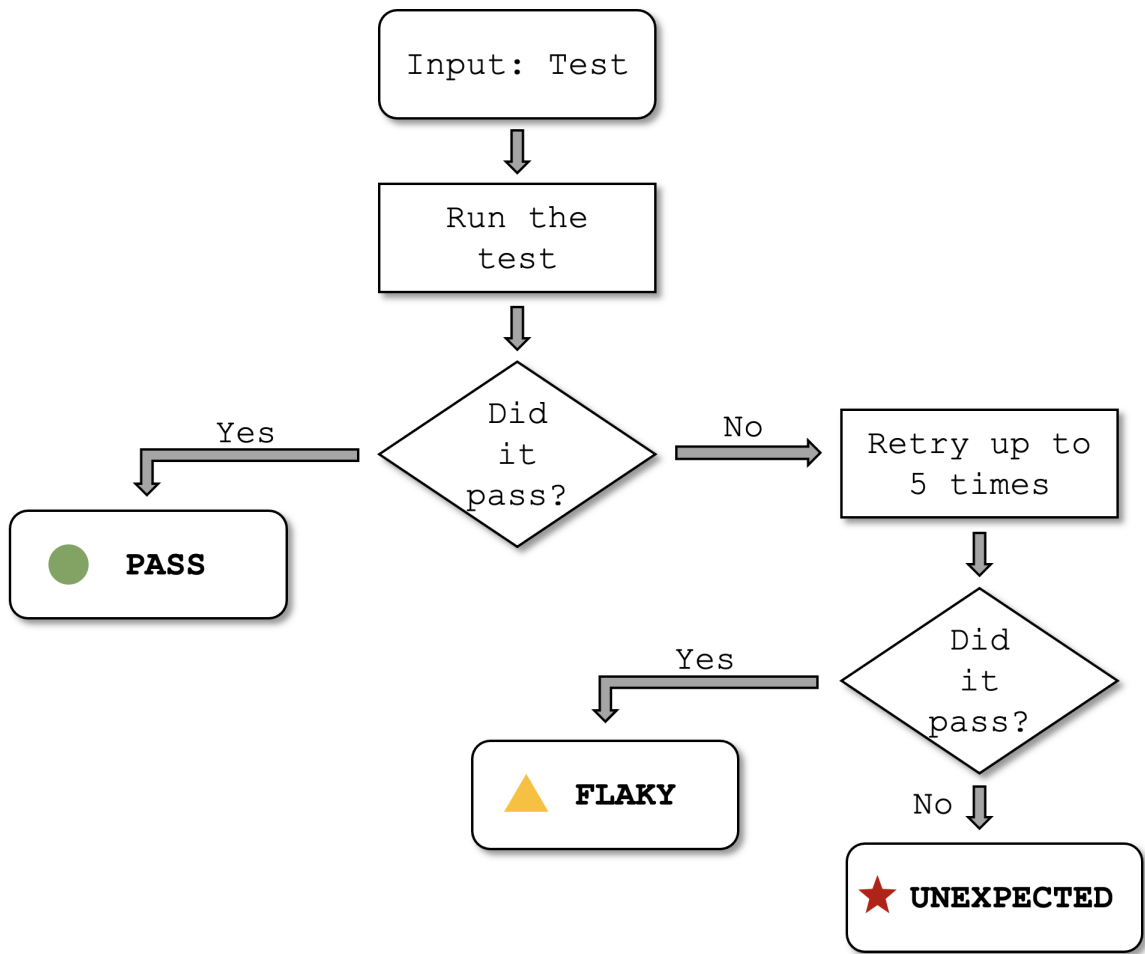


Figure 3.4: Decision tree representing how test outcomes are determined in a build by the Chromium CI. PASS depicts successful tests, FLAKY depicts tests that passed after failing at least once, while UNEXPECTED depicts tests that persistently failed [10].

tests on CI cycles for Blink LayoutTests while the tests are being investigated. On average, every day, 10 tests (min 0 – max 583) are temporarily added to the *testExpectations* until they are fixed. For example, The following line is an example of a test added to this file which allows the test to fail or pass on Mac11-arm64 OS.

```
+crbug.com/1302856 [ Mac11-arm64 ] virtual/exotic-color-space/images/rgb-png-with-cmyk-color-profile.html [ Failure Pass ]
```

By labeling a test as [pass failure] the test will only be run once and will be categorized as “Expected”.

Five possible expected outcomes can be manually assigned to a test:

Pass: a test passes when its output matches the pre-defined expected results. By default, tests are expected to pass. If a test is not named in the *testExpectations* file, then only Pass is an expected outcome.

Failure: a test fails when the outcome of the test does not match the expected result defined as a file that resides next to the test file in the directory.

Skip: Tests marked as Skip will not be run by default, generally because they cause some intractable tool error. For example, Windows-specific tests are skipped on Linux or Mac builds.

Timeout: Tests that take longer than a certain amount of time to complete are aborted and marked as “timed out”. For example, some tests can trigger significant memory allocation in the GPU process, leading to intermittent Out-Of-Memory (OOM) errors and GPU process crashes which yield a TIMEOUT outcome. Tests that commonly time out can be labeled and added to the *testExpectations* file to reduce re-runs.

Crash: Tests that cause the test shell or browser to crash.

As another example, the following line has been added to the file to prevent crashing until it gets fixed. For this change, developers added the following description:

“This CL introduces a change to the maxframes.https.html test that sets the ‘src’ attribute on a fenced frame element that is created in a page that has surpassed the subframe limit. The renderer crashes in this case, so this CL adds TestExpectations to cater to that until the problem is fixed.”¹

¹<https://chromium-review.googlesource.com/c/chromium/src/+3501941>

```
+crbug.com/1123606 virtual/fenced-frame-mparch  
/wpt_internal/fenced_frame/maxframes.https.html [ Crash ]
```

We mine the entire history of tests added and removed to the *testExpectations* file from November 2018 to March 2022 to understand how flakiness has affected Chrome over time. During this time 78,452 changes were made to this file which included adding new tests, removing old tests, and changing the expectation for a test. Even though the flaky tests do not stop the build, the tests are examined in the Chromium Gerrit, Chromium code review system, and a bug report is created for the flaky tests.

3.3 Metrics

Matthew’s correlation coefficient (MCC) is a measure of the quality of binary classification models, particularly when dealing with imbalanced datasets. It was introduced by Brian W. Matthews in 1975. MCC ranges between -1 and +1, with +1 representing a perfect prediction, 0 indicating a random prediction, and -1 indicating a complete disagreement between the prediction and the actual labels.

The formula for calculating MCC is as follows:

$$\text{MCC} = \frac{\text{TP} \times \text{TN} - \text{FP} \times \text{FN}}{\sqrt{(\text{TP} + \text{FP}) \times (\text{TP} + \text{FN}) \times (\text{TN} + \text{FP}) \times (\text{TN} + \text{FN})}}$$

In this formula, TP represents true positives, TN represents true negatives, FP represents false positives, and FN represents false negatives.

3.4 Research Questions

This section embarks on a comprehensive exploration of the research questions aimed at unraveling the complexities surrounding flaky tests in the Chrome ecosystem. The primary objective is to gain a profound understanding of the nature and criteria defining flakiness within the context of Chrome testing. Research Question 1 (RQ1), **“What is the frequency of each category of flaky test?”** drives our exploration to understand the contents of the *testExpectations* file and the

distribution of test categories. This comprehension forms the foundation of our research, guiding us in accurately framing the problem.

A temporal dimension is introduced through Research Question 2 (RQ2), “**How has the number of tracked flaky tests changed over time?**”. This research question focuses on the behavior of tracked flaky tests over time. Understanding the temporal evolution of flakiness is crucial for identifying patterns, trends, and potential contributing factors to changes in the number of flaky tests.

Research Question 3 (RQ3), “**How accurately can we predict the lifetime of a flaky test based on the description**” benefits from novel methodologies to identify patterns and features in test descriptions that align with flaky test behavior over time. The goal is to develop a predictive model to assist developers in prioritizing tests for investigation and debugging.

To understand these research questions, we use a combination of qualitative and quantitative research methodologies, including text analysis, statistical modeling, and temporal trend analysis. The following parts will give more details about the methods we use to answer each question, making sure we thoroughly explore the research goals.

3.5 Objectives and Methodologies

3.5.1 RQ1: Descriptive Statistics - What is the Frequency of Each Category of Flaky Test?

To fully understand the process of handling flaky tests at Chrome, we first need to look at the information stored in the `testExpectations` file. Research Question 1 (RQ1) aims to enhance the understanding of flaky tests in the Chromium dataset by looking at descriptive statistics. The main goal is to figure out how often each type of flaky test occurs, revealing the different ways these tests show instability. This exploration is crucial for creating a smart and well-informed plan to decide which tests to investigate more deeply.

To answer this question, we initially group together all test outcome that involve a combination of “pass” and any other outcome type under the flaky category. By mining the history of changes

made to the `testExpectations` file, we have access to all the tests that have been added to the file throughout the time. To address this research question, we count the number of tests added to the file, including those that were subsequently removed.

RQ1 is about building solid strategies to handle test instability effectively. The results from this research question will give us useful insights for prioritizing tests, which will further help allocate resources wisely and improve processes in testing system of Chromium.

3.5.2 RQ2: Flakiness Survival: How has the number of tracked flaky tests changed over time?

The primary goal of this research question is to explore the evolution of flaky tests over time, investigating the patterns and trends associated with changes in their numbers. We seek to comprehend how the total count of flaky tests fluctuates as new instances emerge and existing ones are resolved and removed from the file. Answering this question specifies the extent of the flaky test issue in Chrome. For instance, if the trend shows consistent increase, it suggests that Chrome's resources, including developers' time, might not be sufficient to effectively address the challenge of investigating flaky tests.

To answer this research question, we start by arranging tests chronologically. The initial count of flaky tests is determined by the number of tests presented on the earliest day available in the changelog. Then for each subsequent day, we update this count by adding the number of new flaky tests and subtracting the number of fixed flaky tests on that particular day. Eventually, we generate the accumulative chart for the number of existing flaky tests over a period of 3 years.

3.5.3 RQ3: Predicting Test Life Time: How accurately can we predict the lifetime of a flaky test based on description

For our third research question, we strive to develop a predictive model that can forecast the duration of a flaky test's persistence, termed its "lifetime," within Chrome's testing environment. The lifetime of a flaky test is measured from its addition to the test expectation file until its resolution and subsequent removal by developers. This temporal metric serves as a valuable indicator for identifying tests that are likely to be promptly resolved, thereby aiding developers in optimizing

their resource allocation. Tests labeled with a “1” by the model will be suggested to developers as having a high probability of being easily addressed.

To develop our predictive model, we use the entire set of flaky tests as our dataset. We examine the distribution shown in Section 3.6.2, and note that 40% of flaky tests never get fixed. Of the remaining 60%, more than half are fixed within 10 to 15 days. We categorize tests into two classes based on the duration it takes to resolve the flaky test: tests that will be fixed “quickly” and those that will take a long time to fix or will never be fixed, i.e., fixed “slowly”. Our primary goal is to develop a model that distinguishes the “quick” class from the “slow”.

We defined “quick10” to be tests that are fixed in ≤ 10 days, and “quick15” to be tests that are fixed in ≤ 15 days. We developed two models: one to predict tests fixed within 10 days (referred to as “quick10”) and another for tests resolved within 15 days (referred to as “quick15”). In both cases, the categories the model predicts are that a flaky test will be fixed quickly or slowly. The model demonstrating superior performance among the two settings was selected as our final model. Our model comprises a pipeline featuring a column transformer for transforming textual data into vectors and a feature selector that identifies the most significant K features using a Random Forest Classifier.

For representing test descriptions, we employ CountVectorizer to convert texts into a matrix capturing token counts. This “bag-of-words” approach has been widely used in previous vocabulary-based methodologies, [7] [21] [30]. Initially, these vectors contain as many features as there are words within the test descriptions, potentially resulting in a sizable dictionary. To address this, we implement feature selection techniques to limit the dictionary’s size, eliminate irrelevant features, and retain the most informative ones. Feature selection not only reduces model training time but also enhances overall performance and interoperability.

The SelectKBest method [39], utilizing a univariate statistical test with χ^2 , is employed to retain the top k features with the highest scores. Fine-tuning the hyperparameters involves adjusting parameters such as the number of trees in the forest, the sampling strategy for SMOTE, and the number of features to retain. After achieving optimal settings, we retrain the model on the entire training set and evaluate its performance using the holdout dataset. This comprehensive approach ensures the effectiveness of our model in accurately predicting the lifespan of flaky tests based on

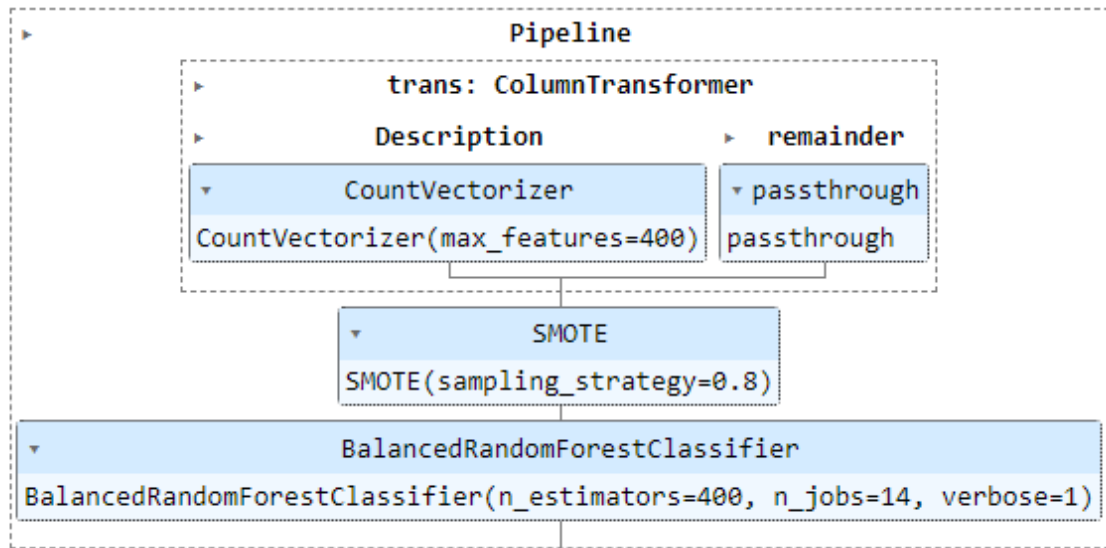


Figure 3.5: Model Architecture

their descriptive features.

Figure 3.5 shows the architecture of the final model after parameter tuning.

3.6 Experimental Results

3.6.1 RQ1: Descriptive Statistics

The goal of this research question is to understand the content of the *testExpectations* file and investigate the distribution of expected outcomes.

What is the frequency of each category of flaky test?

Figure 3.6 demonstrates the frequency of different categories of expected outcomes in the *testExpectations* file. As we can see, “Failure” is the most occurring expected outcome, with 16K tests accounting for 40% of tests in the file. Failures block the build and must be fixed before integrating the change.

The Flaky category is an aggregated group of different test outcomes that include “Pass” and at least one type of failure, including [failure pass], [pass timeout], [failure pass timeout], [crash pass],

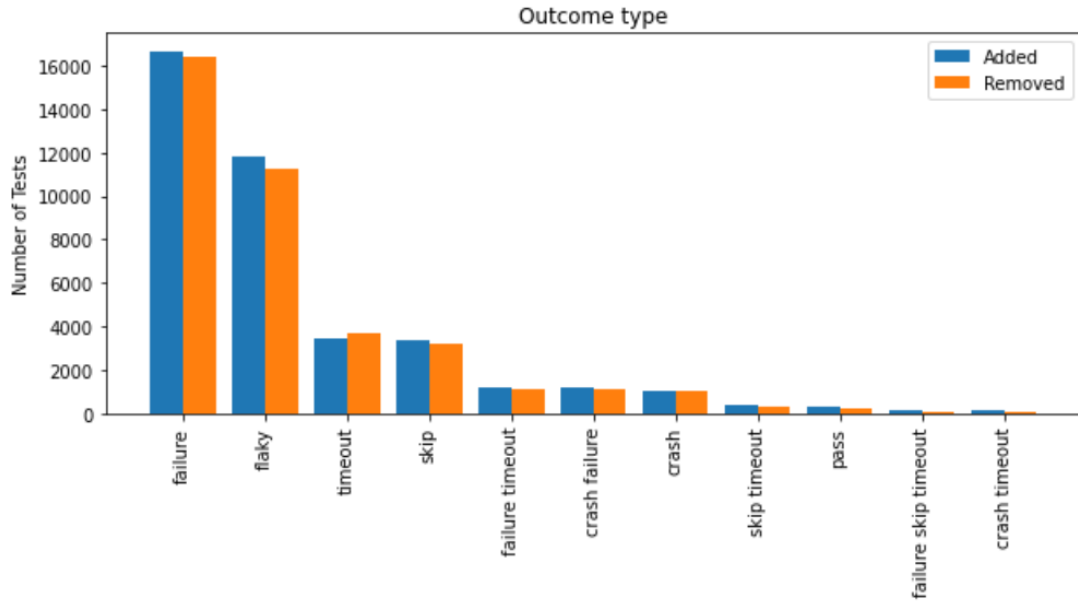


Figure 3.6: Frequency of different outcomes in test expectation file. Failed tests and flaky tests are the most occurring categories with 40% and 30% of the tests in the file accordingly.

etc. In Figure 3.6 we can see that 30% of the added tests to the *testExpectations* file are flaky.

The next common category is “Timeout”. “Timeout” is the expected outcome for the tests that take a long time to finish, so the test runner aborts them. Almost 8% of the tests added to the *testExpectations* file expected to timeout.

When there is a test that developers want to be disabled temporarily, they add it to this file with “Skip” outcome. According to Figure 3.6, 3.3k tests added to the *testExpectations* file with “Skip” as expected outcome.

Table 3.1: Chromium Dataset Summary

Outcome	Count	Bug	BugPercentage	CrossPlatformPercentage
Failure	19,153	13,151	68%	68%
Flaky	12,693	10,306	81%	54%
Skip	4,171	3,881	93%	60%
Timeout	3,597	1,358	37%	25%
Crash	1,089	498	45%	38%

Understanding the Connection Between Flaky Tests and Bug Reports

In our data analysis presented in Table 3.1, we investigate the correlation between test outcomes and bug reports, aiming to understand the impact of flaky tests on the bug reporting scenario. We note that 80% of flaky tests are linked to specific bug IDs, which suppress the connection observed in other test failures. This observation highlights the complexity of addressing the nondeterministic nature of flaky tests, which makes them more challenging to resolve compared to regular failures. As a result, there is an increased number of bug reports, indicating the ongoing effort required to track and resolve these tests over time.

We also explore the dynamics of platform-specific tests, where the expected outcome varies based on the platform. For example, a test might be anticipated to fail only on Windows. Table 3.1 reveals an interesting correlation between the percentage of cross-platform tests and the frequency of assigned bug IDs. This connection suggests that tests demonstrating unexpected behaviors on specific platforms are more likely to attract bug IDs, indicating increased attention from developers. Moreover, our findings indicate that platform-specific failures, where tests exhibit unexpected behaviors exclusively on a particular platform, tend to receive fewer bug IDs. This insight sheds light on a trend where these specific failures may be more frequently overlooked or considered less critical in the development process.

`testExpectations` file is the main test failure suppression method and is used for temporarily marking tests as flaky until they get fixed. The two most occurring categories, “Failure” and “Flaky,” account for a combined total of 70% of all tests.

3.6.2 RQ2: Flakiness Survival: How has the number of tracked flaky tests changed over time?

Figure 3.7 illustrates the cumulative count of flaky tests over 3 years. The graph reveals noticeable spikes and dips on specific dates, indicating substantial fluctuations. These variations align with builds that introduce a considerable number of new flaky tests, attributed to the incorporation

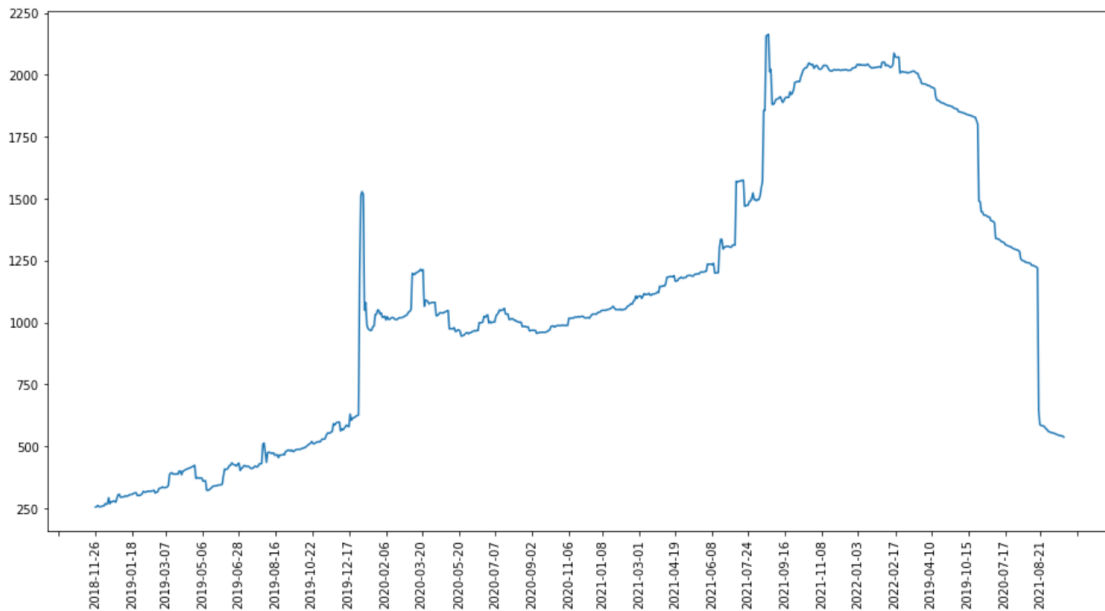


Figure 3.7: Accumulated Number of Flaky Tests Over Time. The steep rise in early January 2020 is due to the visually-refreshed form controls feature activation, which led to the addition of 583 tests to the file until their baseline was updated.

of additional, or removing a group of flaky tests due to enabling or disabling a feature on the platform under study. For example, on January 3, 2020, 583 flaky tests were added to the file because of enabling visually-refreshed form controls on Windows/ChromeOS/Linux. The developers commented, “Because many layout tests use form controls, including tests that are not directly testing form controls, this CL [pull-request] requires around 1200 tests to be rebaselined.” Then they create different bug issues to track the process of cleaning these newly added tests and remove these new lines from TestExpectations.

`testExpectations` file is the main test failure suppression file and is used for temporarily marking tests as flaky until they get fixed. Our investigation reveals a consistent trend: Apart from two notable spikes, there is a gradual rise in the number of flaky tests for the first part of the period. However, towards the end, a significant number of flaky tests are resolved and subsequently removed from the file, leading to a relatively stable count of existing flaky tests. This stability implies that the number of resolved flaky tests is nearly equivalent to the number of newly identified flaky tests introduced over time [40].

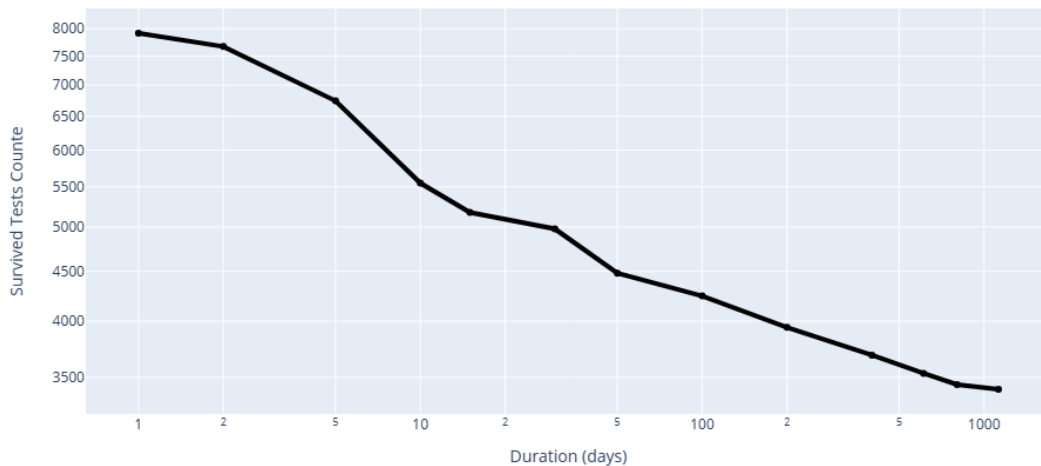


Figure 3.8: Log Scale Survival Chart of Lifetime of Flaky Tests: Approximately 40% of tests get fixed within 15 days. nearly 40% of the tests survive by the end of the period.

Survival Chart of Flaky Tests Lifetime

Figure 3.8 illustrates the survival probability of flaky tests over time. This chart displays the proportion of subjects that have not experienced the event (in our case, the tests that have not been fixed) up to a specific point in time. As time advances, the survival probability diminishes, indicating the proportion of tests that have not survived (i.e., been fixed) by each time point.

Initially, on the first day of observation, we start with a total of 8,279 tests in our dataset. In the initial phase, roughly spanning between 10 to 15 days, there is a noticeable and relatively steep decline in the survival probability. This suggests a higher rate of tests being resolved or fixed within this early timeframe. Upon closer examination, we find that approximately 33% of the tests are resolved before the 10-day mark, with this figure slightly increasing to 38% before 15 days.

Following this initial decline, there is a period where the survival probability levels off at a slower rate. This indicates that tests remaining unresolved beyond a certain point have a higher likelihood of either being left unresolved or being fixed much later. It suggests that these tests may not be considered high priority for developers. Subsequently, there is a secondary, more gradual decline in survival probability starting around the 200th day, signaling that over a more extended

period, additional tests are being resolved.

Towards the end of the observation period, the number of remaining unresolved tests does not reach zero. Instead, it constitutes approximately 40% of all tests, indicating that a substantial portion of tests remain unfixed.

The median survival time, which represents the point at which 50% of the tests have been fixed, falls approximately between 100 and 200 days, with a more precise estimate at around day 122.

In summary, the overall trend demonstrates a decreasing pattern in the number of tests available for debugging, with a more rapid decline observed in the first 15 days, followed by a more stable trend after the 200th day.

Analyzing the evolution of changes in `testExpectations` file over time reveals that except for two short-lived spikes in 3 years, the total number of flaky tests is roughly constant. By studying the survival time of flaky tests, we find that 40% of flaky tests are never fixed. Of the flaky tests that are fixed or removed, 38% are in less than 15 days.

3.6.3 RQ3: Predicting Test Life Time: How accurately can we predict the lifetime of a flaky test based on description?

Outcome measures. We use the standard evaluation metrics precision, recall, F1 score, and Matthews correlation coefficient (MCC) metrics to evaluate the performance of our model and compare different classifiers. These metrics have been used to evaluate the performance of classifiers, including binary classification of flaky tests [7] [21] [26]. Since our dataset is unbalanced, weighted metrics are more suitable for our evaluation.

Training and test data. Our data division strategy involves partitioning the dataset into two segments: The initial 80% of tests constitute our training set, while the remaining 20% form the holdout set. This approach ensures that we maintain the temporal evolution of the text expectation file over time and mitigates any potential data leakage issues that might arise from random selection. Emphasizing this temporal aspect is crucial, as our observations indicate that failing to account for this condition and training a model on a shuffled dataset would lead to a significant overestimation

of performance.

Table 3.2 reports the obtained performance. We achieved a promising precision rate of 73.6% when categorizing tests into two distinct classes: “quick to fix” and “slow to fix”, with “quick to fix” being the tests fixed within 15 days. Within the scope of our research, precision holds paramount significance, as it dictates the proportion of tests labeled for investigation that truly warrant priority attention. A precision rate of 73% signifies that when we recommend a test for investigation, it proves to be genuinely “quick to fix” approximately 73% of the time, ensuring that our suggestions to developers are mostly accurate and avoid unnecessary investigations.

Table 3.2: Result of the model, predicting tests fixed in ≤ 15 days (quick15) and ≤ 10 day (quick10)

Model	Precision	Recall	MCC	F1-Score
quick15	73.6%	57.4%	0.39	64.5%
quick10	62.7%	40%	0.28	49%

Using only the description of a test, we can predict the lifetime of flaky tests with 73% precision and MCC close to 0.4.

3.7 Discussion

3.7.1 Model Optimization Process

To optimize our model, we engaged in parameter tuning, which we will discuss in the following. Table 3.3 presents a comprehensive overview of our predictive model’s performance, highlighting the systematic adjustment and evaluation of various parameters and features. In the following, we delve into a discussion of these outcomes to better understand the model’s behavior:

The base model achieved a decent level of precision (67%) and recall (62%), indicating that it could effectively identify the target variable. The MCC score of 0.34 suggests a moderate level of

Table 3.3: Result of using different configurations and features. The Base Model has 200 estimators and a maximum feature count of 200 and has quick15 as the target class.

Model	Precision	Recall	MCC	F1-Score
Base Model (Max Feature = 200, N estimators = 200)	67%	62%	0.34	0.64
Base Model + Oversampling	67%	63%	0.34	0.64
Base Model + Oversampling + Feature Selection	67%	60%	0.34	0.64
A = Base Model with Expectation and Author features	63%	58%	0.27	0.61
Model A + Feature Selection	66%	62%	0.35	0.64
Base Model (Max Feature = 400)	68.8%	59.3%	0.35	63.7
B: Base Model (Max Feature = 400, N estimators = 400)	73.6%	57.4%	0.35	63.7
Final Model: Model B + oversampling rate = 0.8	73.6%	57.4%	0.39	64.5

agreement between predictions and actual outcomes. The F1 score of 0.64 demonstrates a balanced performance in terms of both precision and recall.

However, the introduction of the Author and Expectation features adversely affected the model's performance, resulting in a decline across all four metrics. Although subsequent Feature Selection implementation showed improvements, it only brought the performance up to the base model's levels, suggesting a limited contribution of these features to overall model performance.

By expanding the maximum number of features to 400, a marginal improvement in precision was noted. Additionally, aligning the number of estimators with the maximum features resulted in a notable precision of 73.6%, indicating an enhanced ability to identify positive cases. These findings imply that a more extensive feature set led to improved model performance, which is reasonable considering the textual nature of our data.

Finally, the application of oversampling resulted in substantial improvements, with the MCC score reaching 0.397 and the F1 score demonstrating a notable increase to 64.5%.

To sum up the optimization process, in the following, we present our major findings:

1. Parameter Tuning (Number of Features and Estimators):

In the first set of experiments, the model was trained with a maximum of 200 features and 200 estimators, accompanied by true self-oversampling at 80%. This configuration yielded the highest precision at 73.6% and a relatively balanced F1 score of 64.5%. However, it's important to note that the recall metric was moderate at 57.4%, indicating a trade-off between precision and recall. The MCC (Matthews Correlation Coefficient) of 0.397 demonstrates a reasonable level of classification performance.

Reducing the number of maximum features to 200 resulted in a slight decrease in precision and F1 score, highlighting the importance of feature selection. This adjustment may help reduce model complexity and potential overfitting.

2. Handling of -1 Values:

Experiments involving the treatment of -1 values showed that dealing with these values directly could improve model performance. Precision and recall generally improved when compared to scenarios where -1 values were not explicitly addressed. This suggests that considering -1 values as a separate category or applying specific imputation techniques can be beneficial for classification tasks.

3. Feature Selection:

Introducing feature selection techniques enhanced model performance consistently. Precision, recall, F1 score, and MCC improved across various experiments. This underscores the importance of identifying and utilizing the most informative features, reducing noise in the data, and potentially mitigating overfitting.

4. Oversampling:

Employing oversampling with a ratio of 0.8, regardless of other parameter settings, generally improved recall but sometimes led to a slight decrease in precision. Oversampling can help address class imbalance, which is often crucial in classification tasks with skewed class distributions. However, it's essential to strike a balance between improving recall and maintaining precision.

5. Additional Features:

Including additional features like "Expectation" and "Author" resulted in improved MCC, F1 score, and recall. These features appear to carry valuable information that enhances the model's

ability to make accurate predictions.

In summary, the performance of the model is highly sensitive to parameter settings, feature selection, and the handling of -1 values. The experimental variations in parameters and features provided valuable insights into model performance. Feature selection appeared to positively impact prediction accuracy, while an increased feature count contributed to higher precision and recall. The combination of oversampling techniques and parameter tuning also led to notable improvements.

Achieving a balanced precision-recall trade-off is crucial, and this can be influenced by the specific problem context and goals. The incorporation of relevant features and appropriate preprocessing techniques can significantly impact model performance, ultimately leading to better predictive accuracy. Further fine-tuning and experimentation may be necessary to optimize the model for specific use cases.

3.7.2 Window Training

To explore the short-term relationship between lifetime and historical data, we implement a window training strategy with two distinct setups. The first configuration utilizes the past six months of data to predict the lifetime class for the subsequent month, while the second configuration reduces the historical window to three months for predicting the following month. To maintain consistency, we employ the same model in both scenarios, adjusting the oversampling rate from 0.8 to 1. The rationale behind this adjustment lies in the reduced number of samples within the training set of the new model. With fewer instances available, the oversampling rate of 0.8 forces the model to move data points from the minority class.

The training process starts from the first day of the dataset, allocating the initial 6 months for training while reserving the 7th month for testing. Subsequently, it assesses the model's performance, stores the outcomes in a list, and advances the training window by one month to iterate the procedure. Ultimately, an average is computed over the list to determine the overall performance of the model.

Table 3.4 summarizes the result of window training models. Although the minimum and maximum values for precision are similar in the two models, by looking at the mode we can see that the model with longer history performed better.

The model achieved an average precision of 59%, ranging from a minimum of 22% to a maximum of 92% across various data windows. Notably, the mode of precision stood at 72.6%, closely aligning with our prediction based on the entire dataset.

Table 3.4: Result of the window training model, for window sizes of 6 months and 3 months

Model	Precision Mean	Precision Mode	Min Precision	Max Precision
6-month window	59.7%	72.6%	22.3%	92.3%
3-month window	53.1%	41.8%	21.7%	93.4%

3.7.3 5-fold Cross Validation

To explore the impact of random dataset splitting, we examined two configurations: one involving a temporal dataset split, where the initial 80% of the data served as the training set, and the remaining portion as the test set, and the other utilizing a 5-fold cross-validation. In the first configuration, we trained our model on 6,600 tests, then, we evaluated it on 1,650 tests. In the second setting, we partitioned the dataset into five equally sized folds. The model is trained on four of the folds and tested on the remaining fifth fold. This process is repeated five times, each time using a different fold as the test set. We achieved 0.65% Precision, 0.25 MCC, 0.48 Recall, and an Accuracy of 0.62% using 5-fold cross-validation, on our best model. This suggests that considering the temporal splitting preserves the nature of data, providing a more realistic evaluation of the model's performance.

3.8 Threats to Validity

3.8.1 Internal Validity.

It is crucial to recognize how human factors can affect the time it takes to fix flaky tests. Changes in development teams and project management practices can impact how quickly issues, including flaky tests, are addressed. Considering and understanding these human-related factors is essential

for predicting fix times accurately. Researchers should be careful in their analysis to understand the effects of these factors, which significantly impact the reliability of predictive models.

3.8.2 External Validity.

When predicting how long it takes to fix flaky tests based on past changes, it is important to consider if the findings apply to different projects and how human factors may influence resolution times. Project variations are significant, and what works for one may not work for others. Researchers need to carefully explore if models from one dataset can be used for different projects, and be transparent about any limitations in generalizing the results.

3.8.3 Construct Validity.

In our investigation, we made the assumption that the removal of a flaky test from the file indicated that developers had addressed the issue, considering it as a sign of successful resolution. However, it's important to acknowledge that in certain cases, test removal could stem from changes in the software's features rather than direct fixes to the test itself. Therefore, the absence of a test in the file doesn't always imply that the flakiness has been effectively addressed.

3.9 Conclusion

While software testing has long acknowledged flakiness as a prominent issue, research on this subject has rarely focused on classifying flaky tests for investigation in a large-scale industrial CI setting. In this research, we collected the history of more than 8K flaky tests on the Chrome project to be utilized for our lifetime classification model. We observe that 40% of the flaky tests remain unresolved. Among the flaky tests that undergo resolution or removal, 33% are addressed within a timeframe of fewer than 10 days. Identifying the appropriate tests to address is crucial, given that we aim to avoid allocating developers' time to the 40% of tests that may prove unsolvable. Our preference lies in pinpointing tests that are both more manageable and hold greater importance for timely resolution.

To effectively prioritize tests for investigation, it is crucial to analyze the distribution of the time

it takes for a flaky test in Chrome to be resolved. This exploration is fundamental for establishing a robust framework to address the underlying issues. By analyzing the distribution of fixing time of flaky tests we observe that except for two brief spikes observed over a 3-year period, our analysis indicates that the overall count of flaky tests remains relatively stable. This trend demonstrates that while developers address flaky tests, new ones continue to arise, maintaining a relatively consistent count of existing flaky tests. This shows the necessity for an efficient test prioritization method to direct developers' efforts towards reducing this trend.

We consider the lifetime of a test as a measure of the effort needed to address the test and its priority for being investigated. By solely analyzing the test description, our model can accurately predict the lifespan of flaky tests, achieving a precision of 73% and an MCC of 0.39. Analyzing test descriptions alone yields a high-precision method for predicting flaky test lifespans, suggesting a practical approach to test management.

Chapter 4

Discerning Legitimate Failures from Flaky Failures

Continuous Integration (CI) is a software engineering process that allows developers to frequently merge their changes in a shared repository [41]. To ensure a fast and efficient collaboration, the CI automates different parts of the development life cycle. Regression testing is an important aspect of CI as it ensures that new changes do not break existing functionality. Test suites are executed for every commit and test result signals whether changes should be integrated into the operational codebase or not.

Tests are an essential part of the CI as they prevent faults from entering the codebase, and they ensure smooth code integration and overall good software function. Unfortunately, as flaky tests, exhibit a non-deterministic behavior, they send false alerts to developers about the state of their applications and the integration of their changes.

Indeed, developers spend time and effort investigating flaky failures, as they can be difficult to reproduce, only to discover that they are false alerts [37]. These false alerts occur frequently in open-source and industrial projects [6], [9], [11], [42] and make developers lose not only time but also their trust in the test signals. This trust issue in turn introduces the risk of ignoring fault-triggering test failures. This way, false alerts defy the purpose of software testing and hinder the flow of the CI.

To deal with test flakiness, many techniques aiming at detecting flaky tests have been introduced. A basic approach is to rerun tests multiple times and observe their outcomes. While to some extent effective, test reruns are extremely expensive [42], [43], and unsafe. To this end, researchers have proposed several approaches relying on static (the test code) [7], [8], [21], [26], [42] or dynamic (test executions) [6], [11], [44] information (or both) [27] to predict whether a given test is flaky. Among the many flakiness prediction methods, the vocabulary-based ones [7] [20] [22] [25] [30] are the most popular [2]. They rely on machine-learning models that predict test flakiness based on the occurrences of source code tokens of the candidate tests. Interestingly, previous research has found these approaches particularly precise, with current state-of-the-art achieving accuracy values higher than 95% [7] [20] [21] [22] [26].

At the same time, vocabulary-based approaches are static and text-based, thus, they are both portable, i.e. limited to a specific language, and interpretable, i.e. users may understand the cause of flakiness based on the keywords that impact the model's decisions. All these characteristics (precision, portability, and interpretability) make vocabulary-based approaches appealing; they are flexible and easy to use in practice. Because of this, we decided to replicate these techniques on an industrial project (the Chromium project) and evaluated their ability to effectively support the detection of flaky tests during the CI operation cycles.

In this chapter, we utilize the dataset provided by Haben et al. [10] to classify the dataset into two categories: flaky failures and legitimate failures. This classification is accomplished through the application of vocabulary-based methods. To define the scope of the problem, our initial step involves understanding the process of labeling flaky tests on Chrome and conducting a comprehensive analysis of the dataset.

The subsequent sections of this chapter are structured as follows. Initially, we provide an overview of the Chrome Continuous Integration (CI) process and elaborate on the characteristics of our dataset. Following this, we offer a concise introduction to the machine learning methodologies employed in our study. Section 4 delves into the original study along with its associated limitations. In Section 5, we outline our research objectives and present the research questions. Subsequently, we present the outcomes of our analysis and address the research questions posed earlier. Lastly, we engage in a comprehensive discussion of the findings and outline potential threats to the validity

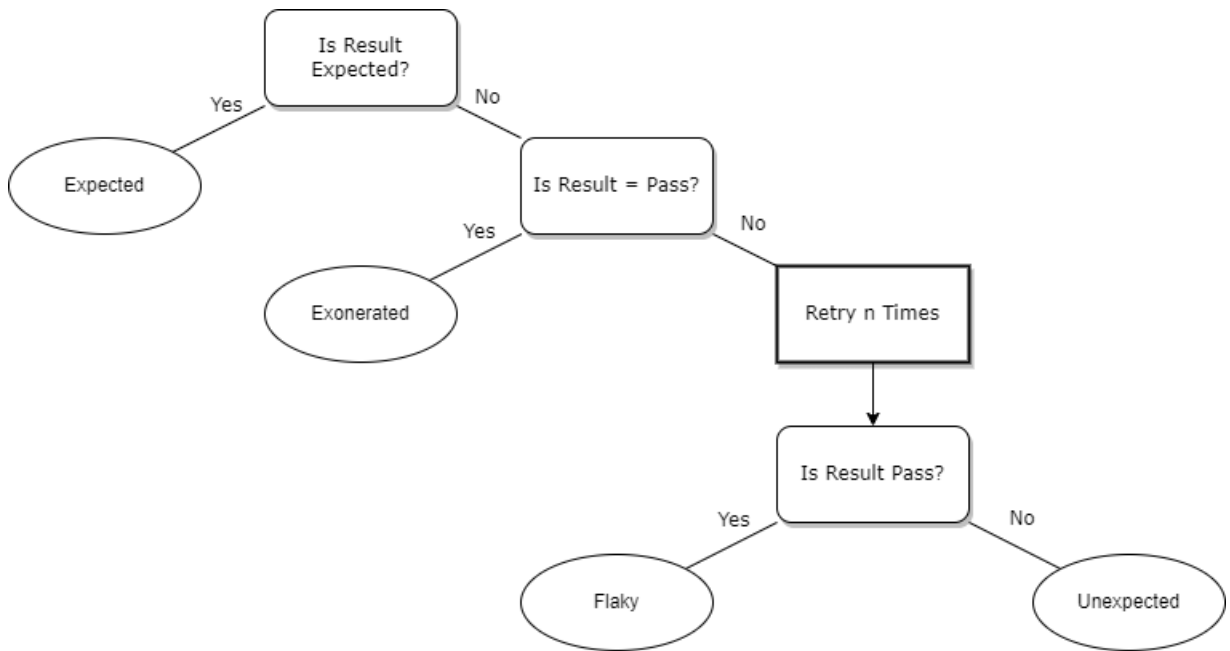


Figure 4.1: Decision tree representing how test outcomes are determined in a build by the Chromium CI. PASS depicts successful tests, FLAKY depicts tests that passed after failing at least once, while UNEXPECTED depicts tests that persistently failed. [38]

of the research. The concluding section compactly summarizes the entirety of the chapter.

4.1 Chrome Process

Chromium employs a distinctive approach to handle flaky tests, as depicted in Figure 4.1. The decision-making process employed by LuCI (Logical Unit of Chromium Infrastructure) determines the outcome of a specific test within a build. A test is deemed successful if it passes without any errors during a single execution. However, in case of failure, LuCI initiates automatic retesting, conducting the test up to five times. If all reruns yield failures, the test is labeled as “unexpected”, resulting in a build failure. In the original study, such unexpected tests are referred to as “fault-revealing” tests.

Moreover, if a test manages to succeed despite encountering one or more failed executions within the same build, it is classified as “Flaky”. However, this classification does not hinder the overall progression of the build, as it still proceeds as though there had been no failure.

Figure 4.1 shows how tests are grouped into four categories, and we discuss the process below.

Expected. Tests can have five outcomes: pass, fail, abort, timeout and crash. Developers can specify the acceptable outcomes in the `testExpectations` file. If no outcome is listed in the file, then the only acceptable outcome is a pass. Consequently, when the test execution result aligns with the expected outcome, the test is labeled as “Expected”.

Exonerated. When a test is expected to fail but passes, the test has an unexpected outcome, but is categorized as Exonerated because tests are always initially designed to pass. Developers can optionally check the list of tests that were expected to fail but passed. In the figure 4.2 “`compositing/gestures/gesture-tapHighlight-2-iframe-scrolled-inner.html`” is an Exonerated test because it passed even though it was expected to fail.

Flaky. Tests that fail unexpectedly are re-run multiple times. If one of the re-runs passes, the test is categorized as Flaky, and the build can still be integrated.

Unexpected. And Finally, if a test fails and none of the reruns pass, the test will be labeled as “Unexpected”. Unexpected tests trigger a build failure. In this study, unexpected tests are referred to as fault-revealing tests.

Figure 4.2 provides insight into a typical build result in the Chromium project. In this build, one test exhibited an unexpected outcome, prompting 4 subsequent reruns to verify the outcome’s reliability. We also can see that 212 tests were identified as flaky, highlighting instances where tests transitioned from initial failures to subsequent passes. Moreover, the results reveal that 233 tests passed despite their expected failure status, resulting in their classification as exonerated tests.

4.2 Data

In this chapter, we employ two datasets for our study. The initial dataset that is used for research question 1 (RQ1) comprises the outcomes of all test runs on the Linux builder throughout January 2022. The second dataset is a subset of the initial dataset over a longer period and is used to answer RQ2 and RQ3. This difference is motivated by the fact that gathering all the tests for nine months would not be practical due to its substantial size. Table 4.1 provides statistical insights into the extent of the flaky test challenge within the Chromium project. In the span of just one month,

Build [chromium](#) / try / [mac10.13-blink-rel](#) / 14948 | [Switch to the legacy build page](#)

Overview **Test Results 1** Steps & Logs Related Builds Timeline Blamelist

[Configure Table](#)

S	Name
▼	1 test variant: status= <i>UNEXPECTED</i> , test_suite= <i>blink_web_tests</i>
▼	virtual/fenced-frame-mparch/wpt_internal/fenced_frame/pointer-lock.https.html history source ID: <code>ninja://:blink_web_tests/virtual/fenced-fra...</code> builder: <code>mac10.13-blink-rel</code> , test_sui <ul style="list-style-type: none"> > 3.4s run #1 <i>unexpectedly failed</i> in task: 5793d68cc215be11 > 3.2s run #2 <i>unexpectedly failed</i> in task: 5793d68cc215be11 > 2.8s run #3 <i>unexpectedly failed</i> in task: 5793d68cc215be11 > 2.3s run #4 <i>unexpectedly failed</i> in task: 5793d68cc215be11
▼	212 test variants: status= <i>FLAKY</i> , test_suite= <i>blink_web_tests</i>
>	<code>css3/filters/backdrop-filter-boundary.html</code>
▼	editing/pasteboard/data-transfer-items-image-png.html history source ID: <code>ninja://:blink_web_tests/editing/pasteboa...</code> builder: <code>mac10.13-blink-rel</code> , test_sui <ul style="list-style-type: none"> > 423ms run #1 <i>unexpectedly failed</i> in task: 5793d689fd3d8711 > 2.7s run #2 <i>expectedly passed</i> in task: 5793d689fd3d8711
>	<code>wpt_internal/css/selectors/focus-visible-select-001.html</code>
▼	233 test variants: status= <i>EXONERATED</i> , test_suite= <i>blink_web_tests</i>
▼	compositing/gestures/gesture-tapHighlight-2-iframe-scrolled-inner.html history source ID: <code>ninja://:blink_web_tests/compositing/ges...</code> builder: <code>mac10.13-blink-rel</code> , test_sui Unexpected passes are exonerated <ul style="list-style-type: none"> ▼ 341ms run #1 <i>unexpectedly passed</i> in task: 5793d698052a4111 <ul style="list-style-type: none"> > Tags: <code>monorail_component: Blink>Compositing</code>, <code>raw_typ_expectation: Failure</code>, <code>step_name: bli</code>
>	<code>css3/blending/background-blend-mode-single-accelerated-element.html</code>
>	<code>wpt_internal/bluetooth/requestDevice/filter-does-not-match.https.html</code>
▼	0 test variants: status= <i>EXPECTED</i> , note: custom grouping doesn't apply to expected tests

Figure 4.2: Sample Build Result in Chromium. This build had 212 flaky tests 233 exonerated and one unexpected test.

Chrome executed an immense volume of tests, totaling 27 million conducted across 99K distinct tests. Among these test runs 820K instances resulted in test failures, highlighting the prevalence of issues encountered during testing. Remarkably, a substantial portion of these failures, totaling 66,189, were identified as flaky test runs, indicating tests that exhibit inconsistent behavior across multiple executions. These flaky tests were associated with 970 distinct test cases, reflecting the diversity of scenarios where such unpredictability arises. Within these builds, a notable subset of 21,932 test cases were identified as contributing to flaky failures, underscoring the need for targeted strategies to address these specific issues. These figures vividly illustrate the significant scale of the flaky test issue within the Chromium project.

It is important to note that this dataset is utilized solely for illustrating the considerable scale of flaky test occurrences in the Chrome environment. Subsequently, for the remainder of this paper, we exclusively utilize the dataset introduced by Haben et al. [10]. This choice is motivated by our objective to gather features such as test sources and test logs over an extended duration. The entire set of tests within the Chrome project would be impractical to work with due to its substantial size. Consequently, tests lacking essential features or those influenced by infrastructure issues are omitted from our analysis to ensure a more manageable dataset for the proposed research.

Table 4.1: Chromium Dataset Summary for one month (January 2022)

Number of test runs	27,959,773
Number of distinct tests	99,723
Number of failed test runs	819,572
Number of builds	231
Number of flaky test runs	66,189
Number of distinct flaky tests	970
Number of builds with flaky failure	231
Number of flaky failure tests	21,932

Table 4.2 presents an in-depth analysis of the frequency distribution for each final status category. Notably, the count of flaky tests in this table closely mirrors the figures previously outlined in Table 4.1. However, it's essential to clarify the inconsistency observed between the statistics

presented in Tables 4.2 and 4.1 concerning failed tests. This difference arises from the classification criteria employed, particularly in cases where tests encounter failures and undergo multiple reruns. When a test fails and is subjected to four subsequent reruns, it's classified as an unexpected outcome, leading to a count of one unexpected outcome and five test failures. Additionally, it's worth noting that a subset of failed tests contributes to the category associated with flaky failures. An important observation to highlight is that not every instance of test failure automatically results in the test being labeled as unexpected.

Table 4.2: Number of Tests for each category of Final Status

Final Status	Number of tests
Flaky	66,189
Exonerated	33,563
Unexpected	11,856

Table 4.3 compares flaky test occurrence to the whole dataset. Notably, flaky failures represent only 2.6% of the total failures.

Table 4.3: Flakiness magnitude

Percentage of failures that are flaky	2.6%
Percentage of test runs that are flaky	0.23%
Percentage of unique tests that are flaky	0.9%

4.2.1 Dataset Features

Table 4.4 provides a comprehensive overview of key features associated with test executions in the Chrome testing Environment.

buildId: The unique build number is linked to the respective test execution, providing a reference point for tracking and associating tests with specific builds.

flakeRate: The flake rate of the test over the last 35 builds, indicating the proportion of instances where the test exhibited flakiness. This metric aids in assessing the stability of the test over time.

runDuration: The duration of the test execution, represents the time taken for the test to complete its run. This feature is crucial for analyzing the efficiency and performance of individual tests.

runStatus: A categorical feature capturing the status of the test execution. Possible values include ABORT, FAIL, PASS, CRASH, and SKIP, providing insights into the outcome of each test run.

runTagStatus: Another categorical feature denotes additional information about the test execution. Possible values encompass CRASH, PASS, FAIL, TIMEOUT, SUCCESS, FAILURE, FAILURE ON EXIT, NOT RUN, SKIP, and UNKNOWN, offering a more detailed characterization of the test's status.

testSource: The source code associated with the test, facilitates a direct link to the codebase and aids in debugging and analysis.

testSuite: The test suite to which the test belongs, providing context and grouping tests based on their functional or logical associations.

testId: The unique identifier for the test, representing its name. This feature allows for individual test tracking and identification.

4.3 Background

Over the past few years, there have been significant advancements in machine-learning methods applied to text analysis tasks and Natural Language Processing (NLP) models. In this section, we mention three approaches that we utilize to solve the problem of flaky failure classification.

4.3.1 Bag of Words

In Natural Language Processing, the “Bag of Words” (BoW) model stands out as a fundamental and influential concept. This model, rooted in linguistic theories, has played a pivotal role in shaping the landscape of text representation and analysis. The foundational idea behind the Bag of Words model is to treat a document as an unordered collection of words, dismissing the word order and grammatical structure. Instead, it focuses solely on the occurrence and frequency of individual words within a document. This abstraction facilitates a simplified yet effective representation of

Table 4.4: Description of our features. Column Feature Name specifies the identifiers used in our dataset, while Column Feature Description details the features

Feature Name	Feature Description
buildId	The build number associated with the test execution
flakeRate	The flake rate of the test over the last 35 builds
runDuration	The time spent for this test execution
runStatus	<p>ABORT</p> <p>FAIL</p> <p>PASS</p> <p>CRASH</p> <p>SKIP</p>
runTagStatus	<p>CRASH</p> <p>PASS</p> <p>FAIL</p> <p>TIMEOUT</p> <p>SUCCESS</p> <p>FAILURE</p> <p>FAILURE_ON_EXIT</p> <p>NOTRUN</p> <p>SKIP</p> <p>UNKNOWN</p>
testSource	The test source code
testSuite	The test suite the test belongs to
testId	The test name

textual data, offering a numerical and quantitative basis for further analysis.

In its computational manifestation, the BoW model involves constructing a “bag” or a vector representation for each document, where the dimensions of the vector correspond to the unique

words present in the entire corpus. The value in each dimension reflects the frequency of the corresponding word in the document. Consequently, each document is represented as a high-dimensional vector, and the entire corpus forms a matrix where each row corresponds to a document.

4.3.2 Word2Vec

Word2Vec is a popular natural language processing (NLP) technique that embeds words into continuous vector spaces, capturing semantic relationships between words based on their contextual usage. Developed by Mikolov et al. [45], Word2Vec represents words as dense vectors in a high-dimensional space, where the distances and directions between vectors reflect the similarity and analogy relationships between corresponding words. The model operates on the distributional hypothesis, assuming that words with similar meanings tend to appear in similar contexts. The resulting embeddings can be utilized for various NLP tasks such as sentiment analysis, machine translation, and information retrieval.

In the context of our study, we leverage Word2Vec as part of our flaky test prediction methodology. By incorporating semantic information encoded in word embeddings, we aim to enhance the understanding of the textual features associated with flaky tests. The model's ability to capture subtle relationships between words enables a more contextually informed analysis of test-related text, potentially improving the accuracy of our flakiness prediction. We build upon previous studies that have successfully applied Word2Vec in software engineering contexts [7, 10, 20], adapting its principles to the specific challenges posed by identifying and mitigating flaky tests in software development environments.

4.3.3 BERT Model

Devlin. et al. [46] introduced Bidirectional Encoder Representations from Transformers (BERT), a language model aiming to pre-train deep bidirectional representations from unlabeled text. BERT achieves this by conditioning on both the left and right context in all layers. The pre-trained BERT model can then be fine-tuned with only one additional output layer to achieve state-of-the-art performance in various tasks, including question answering and language inference, without requiring significant task-specific architecture adjustments.

Through masked language modeling, BERT learns to predict missing words in a sentence, considering the context of the surrounding words. This process equips BERT with a deep understanding of the relationships between words and their contextual meanings. Fine-tuning on specific tasks, such as sentiment analysis or named entity recognition, allows BERT to adapt its learned representations to perform well on a wide array of natural language processing tasks. BERT's ability to capture context, semantics, and relationships between words has led to its widespread adoption and success in various language-related applications.

To be sure we are covering a variety of ML approaches with different levels of model complexity and architecture, we apply a pre-trained Bert model to our dataset. Specifically, we adopt DistilBERT, a lightweight variant of BERT to classify test failures into two groups legitimate failures and flaky failures. Our approach revolves around a text classification model architecture, where we leverage the DistilBERT backbone pre-trained on extensive textual data to capture complex patterns within test source code. The process begins by tokenizing the input text using the DistilBERT tokenizer, which generates a numerical representation that maintains the contextual information of words in the form of word embeddings. During the training process, the model is fine-tuned using a cross-entropy loss function. The loss is computed by comparing the predicted labels against the actual ones. This supervised learning approach allows the model to iteratively adjust its parameters to minimize the prediction error. To assess the model's performance, we utilize standard metrics such as precision, recall, F1 score, and accuracy. These metrics provide a comprehensive evaluation of the model's performance. The implementation leverages the PyTorch library, which provides a flexible and efficient platform for deep learning. Additionally, the transformers library is utilized for seamlessly integrating the DistilBERT model and tokenizer into the architecture. The model is trained over multiple epochs, each epoch involving a pass through the entire training dataset. Throughout the training process, we monitor the model's loss and performance on validation data. The model's robustness and generalization capabilities are crucial aspects evaluated during this phase.

4.4 Original Study

In the second chapter, we discovered the effectiveness of vocabulary-based techniques in predicting test flakiness. These techniques leverage machine learning to estimate the probability of a test demonstrating flakiness by analyzing textual characteristics in the code. Remarkably, recent progress has boosted the accuracy of these techniques to surpass 95%. Their simplicity and emphasis on textual analysis enable them to seamlessly integrate with specific programming languages and are easily comprehensible. Their popularity stems from their precision, compatibility with various languages, and practical usability.

On the other hand, the majority of studies on flaky tests focus on predicting flaky tests, and the problem of predicting flaky test failures and discriminating between fault-triggering and flaky failures has largely been ignored by previous research. To address the limitations of current methods, Haben et.al [10] focus on classifying test failures into false alerts and legitimate failures. The authors reported that their approach can accurately distinguish legitimate failures from false alerts, achieving a Matthews Correlation Coefficient (MCC) of up to 95%.

The study done by Haben et.al [10] highlights an important aspect of flaky tests and provides a comprehensive dataset, which serves as a foundational resource for further exploration and analysis in this field. However, it is essential to recognize and tackle certain inherent limitations within their methodology. Upon examination of their proposed model, a notable vulnerability emerges, stemming from the random allocation of data into training and testing sets. While such an approach is widespread in machine learning, it exposes the study to the risk of data leakage, a phenomenon wherein information from the testing set inadvertently influences the training process, thereby compromising the integrity and reliability of the results. Our hypothesis suggests that the performance of the model will vary when employing a time-sensitive approach to splitting the dataset for training and testing. This limitation provided valuable insights into areas of improvement and refinement for our research. Overall, the strengths and constraints of this study motivated us to develop our research based on its foundation.

4.5 Objectives and Methodologies

4.5.1 RQ1:How often does each test fail without identifying a fault?

In our study, the primary focus revolves around identifying false alert test failures and constructing a framework to predict such occurrences. However, before delving into the development of predictive models, it is crucial to understand the frequency of this phenomenon. To achieve this, we embark on an analysis of a 30-day dataset of test results, where our objective is to identify unique tests based on their distinctive identifiers and ascertain the occurrences of flaky failures. By investigating how frequently individual tests fail without indicating a fault, we aim to gain insights into the extent of flakiness within the testing environment. This analysis lays the groundwork for identifying patterns and trends in test behavior, which, in turn, will facilitate the development of strategies to mitigate the impact of flaky failures on software test quality.

The flake rate is determined by dividing the number of instances where a test exhibited flakiness by the total number of test runs. To accomplish this, we first identify unique tests based on their unique identifiers, counting those that experienced at least one flaky failure. For each of these unique tests, we compute the corresponding flake rate.

We initiate our exploration of this research question by determining the count of unique tests within a 30-day dataset of test results. Among the 99,000 distinct tests, we identified 970 unique tests that experienced at least one instance of a flaky failure, which accounts for approximately one percent of all unique tests. For each test result associated with a test Id we compute the flaky rate using the following methodology:

$$FlakeRate = \frac{NumberOfFlakyFailures}{NumberOfAllTheTestRuns} \quad (2)$$

The outcome reveals that 50% of flaky tests exhibit a flake rate lower than 0.09, indicating that half of the flaky tests experience flakiness in less than 9% of their runs.

Additionally, we calculate the flip rate, as introduced in [9], for each version of a test. The flip rate serves as a metric to approximate the degree of flakiness exhibited by a test. In our dataset,

84% of flaky tests exhibit a flip rate exceeding 0.8, signifying that a great majority of tests frequently transition between two possible outcomes, indicating a high degree of flakiness.

4.5.2 RQ2: Can we propose a model to discern legitimate failures from flaky tests?

In addressing the following research questions, we work on the dataset provided by Haben et al. [10]. This study is the first to delve into the Chromium dataset, offering a practical perspective on flaky tests within an industrial context. The researchers gathered test execution data from 10,000 consecutive builds facilitated by the Linux Tester, utilizing queries to the LuCI API. This dataset spans nine months from March 2022 to December 2022. The study by Haben et al. [10] provides a compelling exploration of flaky tests in a large-scale industrial setting, addressing real-world challenges. The significance of flaky tests becomes evident when examining the monthly test runs of Chromium, where millions of tests are conducted, revealing a substantial occurrence of thousands of flaky tests. This observation contrasts with prior works like [9], which predominantly relies on synthetic data.

In prior investigations [20] [21] [22] a predictive analysis of test flakiness has been conducted utilizing the source code of the tests and the code under examination. However, a limitation in the majority of these studies lies in their application to relatively small datasets or their lack of evaluation within the context of a continuous integration process, thereby diverging from real-world scenarios. Consequently, we recognized the need to explore the applicability of using test sources for predicting flakiness in a more practical context.

In this chapter, we apply state-of-the-art flakiness prediction methods to the Chromium dataset and check their performance. Our investigation initiates with a straightforward bag-of-words and decision tree methodology, progressively transitioning to more sophisticated approaches like Word2Vec and the BERT language model. This comprehensive exploration includes a range of analytical strategies to enhance our understanding of flakiness prediction in the context of large-scale test runs.

Bag of Words with Random Forest. In our initial methodology, we employ the bag of words

technique to transform the textual data (test source code) into a vector, while utilizing OneHotEncoder to convert the categorical feature (test suite) into a vector representation. The remaining features, such as flake rate and run duration, are numerical and are directly input into the model. Subsequently, a random forest classifier is employed to categorize tests into two groups: flaky failures and legitimate failures. In our pursuit of optimal performance, we explore additional techniques, including oversampling and feature selection methods, the outcomes of which will be discussed in the subsequent section.

Word2Vec with Weighted Logistic Regression. For our second method, we use Word2Vec to convert textual data to vectors. We start our setting with a vector size of 100, an initial learning rate of 0.025, and a min count equal to 2, which ignores all words with a total frequency lower than two. Then we used a weighted Logistic Regression to reward the model `minority_class_weight` times more when it predicts a sample of the minority class correctly. We use this setting because almost 98% of our samples belong to the flaky tests class, which means if the model classifies all the samples to class label 0 (flaky tests), it will achieve a 98% accuracy, with a very high false positive rate. With the weighted model, we encourage the model to classify more samples into the target class which is the samples with label 1 (legitimate failures).

BERT. In our pursuit of exploring diverse model types with varying complexities and methods for processing textual data, we introduce BERT, a groundbreaking natural language processing model that has significantly impacted the field of language understanding. BERT, short for Bidirectional Encoder Representations, leverages a transformer architecture, a neural network design known for its proficiency in capturing intricate patterns and dependencies within data. The distinguishing feature of BERT lies in its bidirectional approach, enabling it to simultaneously consider both the left and right context of a word, fostering a more holistic comprehension of the context in which words are situated.

At the core of BERT's functionality is its pre-training phase on vast corpora of text data, a process that equips it with the ability to acquire contextualized embeddings for words. This pre-trained BERT model can then be fine-tuned with the addition of only one extra output layer, showcasing its

adaptability and efficiency in achieving state-of-the-art performance across various language-related tasks. This approach ensures a robust exploration of different model complexities and textual data processing methodologies, with BERT standing out as a versatile and powerful candidate in our experimentation.

Figure 4.3 illustrates the architecture of the proposed model. As we can see It utilizes a DistilBERT model as its backbone, which is a variant of the BERT model specifically optimized for lightweight and efficient performance. The DistilBERT model comprises several key components, including embeddings, transformers, and classification layers.

The DistilBERT model comprises several key components, including embeddings, transformers, and classification layers.

Embeddings: These components are responsible for converting input tokens into numerical representations called embeddings. The word embeddings capture the semantic meaning of words, while the position embeddings encode the positional information of tokens within the input sequence. These embeddings are processed through layer normalization and dropout operations to enhance model performance and prevent overfitting.

Transformers: The transformer architecture is central to the DistilBERT model and consists of multiple transformer blocks. Each transformer block contains self-attention mechanisms that allow the model to weigh the importance of different words in the input sequence when making predictions. These self-attention mechanisms are complemented by feed-forward neural networks (FFN) and GELU activation functions, which facilitate non-linear transformations and feature extraction from the input data.

Classifier: At the output layer, a linear classification layer is applied to the representations learned by the DistilBERT backbone. This layer maps the high-dimensional feature vectors produced by the transformer blocks to the desired output classes, in this case, binary classes indicating whether a test is flaky or not.

4.5.3 RQ3: Are some test suites easier to predict than others?

In the context of Research Question 3, our primary objective is to find out if the proposed model performs better on a subset of data. This involves a detailed exploration of the flakiness


```

FlakyTestClassifier(
  (back_bone): DistilBertModel(
    (embeddings): Embeddings(
      (word_embeddings): Embedding(30522, 768, padding_idx=0)
      (position_embeddings): Embedding(512, 768)
      (LayerNorm): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
      (dropout): Dropout(p=0.1, inplace=False)
    )
    (transformer): Transformer(
      (layer): ModuleList(
        (0-5): 6 x TransformerBlock(
          (attention): MultiHeadSelfAttention(
            (dropout): Dropout(p=0.1, inplace=False)
            (q_lin): Linear(in_features=768, out_features=768, bias=True)
            (k_lin): Linear(in_features=768, out_features=768, bias=True)
            (v_lin): Linear(in_features=768, out_features=768, bias=True)
            (out_lin): Linear(in_features=768, out_features=768, bias=True)
          )
          (sa_layer_norm): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
          (ffn): FFN(
            (dropout): Dropout(p=0.1, inplace=False)
            (lin1): Linear(in_features=768, out_features=3072, bias=True)
            (lin2): Linear(in_features=3072, out_features=768, bias=True)
            (activation): GELUActivation()
          )
          (output_layer_norm): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
        )
      )
    )
  )
  (classifier): Linear(in_features=768, out_features=2, bias=True)
)

```

Figure 4.3: Architecture of model developed based on Bert

characteristics inherent in different test suit categories, specifically investigating the potential for predicting faults from flaky failures within each category. Additionally, we examine the distribution of flake rates among tests demonstrating flaky behavior compared to those encountering legitimate failures across various categories. The goal is to identify an optimal threshold for classifying tests into the categories of flaky or legitimate failure, based on flake rates.

We classify the tests based on the test suit into three categories: Layout Tests, Browser Tests, and Unit Tests.

Layout tests are used by Blink (browser engine) to test many components, including but not limited to layout and rendering. Typically, these tests include loading pages in a test renderer and then comparing the resulting output, either rendered content or JavaScript output, with an anticipated output file [47].

Browser Tests designed specifically for integration testing of Chrome features within the browser process. The tests run after the browser process initializes and a window is created. To prevent interference among tests, each test operates within its distinct browser process. Browser tests involve the launch of a complete browser and subsequently execute the test within this instance. Typically, files housing browser tests are identifiable by the suffix `_browsertest.cc` [48] [49].

Unit Tests provide a mechanism for testing small sections of code in isolation from the rest of the extensions, and outside of the browser. These tests aim to verify specific parts of the Chromium codebase within a controlled and isolated testing environment. Typically, unit tests are contained in files denoted by the suffix `_unittest.cc` [48] [50].

4.6 Experimental Results

4.6.1 RQ1: How often does each test fail without identifying a fault?

Addressing this research inquiry starts with quantifying the number of unique tests within a 30-day dataset of test results. Among the 99,000 unique tests, 970 exhibited at least one flaky failure, counting for approximately one percent of all distinct tests.

In Figure 4.4, we depict the cumulative flake rate distribution for the 970 unique tests identified

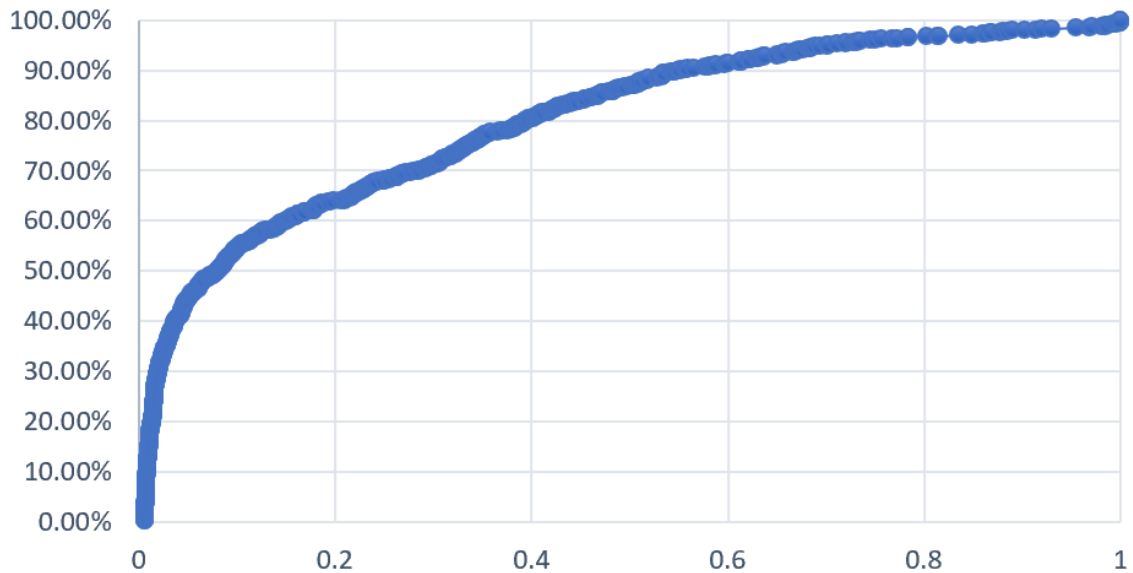


Figure 4.4: Cumulative flake rate for the 970 unique flaky tests

with at least one instance of flaky failure. Notably, over 65% of these tests demonstrate a flake rate below 0.2, indicating that more than half of the identified flaky tests experience flakiness in less than 20% of instances. Moreover, the figure highlights that certain tests exhibit flakiness in more than 70% of instances, emphasizing the significance of this phenomenon. The primary focus lies in the first half of the spectrum, where tests exhibit flakiness only minimally, suggesting that failures in these tests likely arise from code bugs rather than flakiness.

For each unique test ID, we consider every build of that test as a distinct version and compute the number of re-runs for each version. In total, we observed 26,797 different versions that underwent at least one transition from pass to fail or vice versa within the version. In the Chromium project, when there are two different test results for a version, the practice is generally to avoid re-running the test. This results in 66% of tests having a re-run number of one within each version, while 22% have been run again two times, and the maximum number of re-runs is capped at 7.

To capture the temporal variability of test results, we count the number of flips in test outcomes, representing the instances when a test result shifts from failed to pass or vice versa. By calculating the flip rate, as expressed in equation 3, we can determine how frequently flaky tests exhibit flakiness. The flip rate is computed for each version (v) as follows:

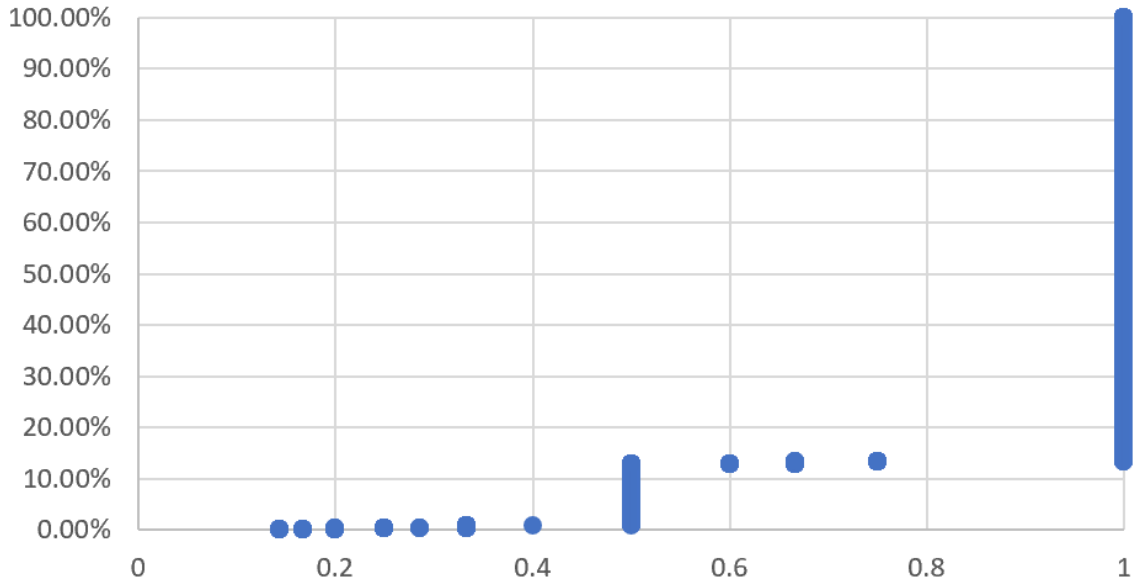


Figure 4.5: Distribution of flip rate values for versions

$$FlipRate_v = \frac{NumberOfFlips_v}{NumberOfPossibleFlips_v} \quad (3)$$

As previously noted, the limited number of runs, typically only 2, for the majority of flaky tests implies an evident outcome – a high portion of versions will have a flip rate of 1. The surprising result of the flip rate calculation for test versions reveals that 86% of versions indeed exhibit a flip rate of 1. This suggests that the flip rate might not be a reliable metric for ranking flaky tests for further investigation. Beyond the fact that 86% of versions share a similar value for this metric, the overall variability in possible values is not substantially high. To be more precise, even within the remaining 14% that do not have a flip rate of 1, 90% exhibit a flip rate of 0.5.

Figure 4.5 illustrates that most versions have a flake rate of 1, leading us to question the efficacy of flip rate as a measure to capture the flakiness level of tests, at least within the Chromium dataset. To validate this assumption, an aggregation of flip rates across all versions of a unique test is necessary. Figure 4.6 presents a cumulative chart showcasing the distribution of flip rates for all tests. According to this chart, the minimum value of the flip rate for a test is 0.5. Furthermore, 84% of tests have a flip rate higher than 0.8, with 44% of tests having a flip rate equal to 1.

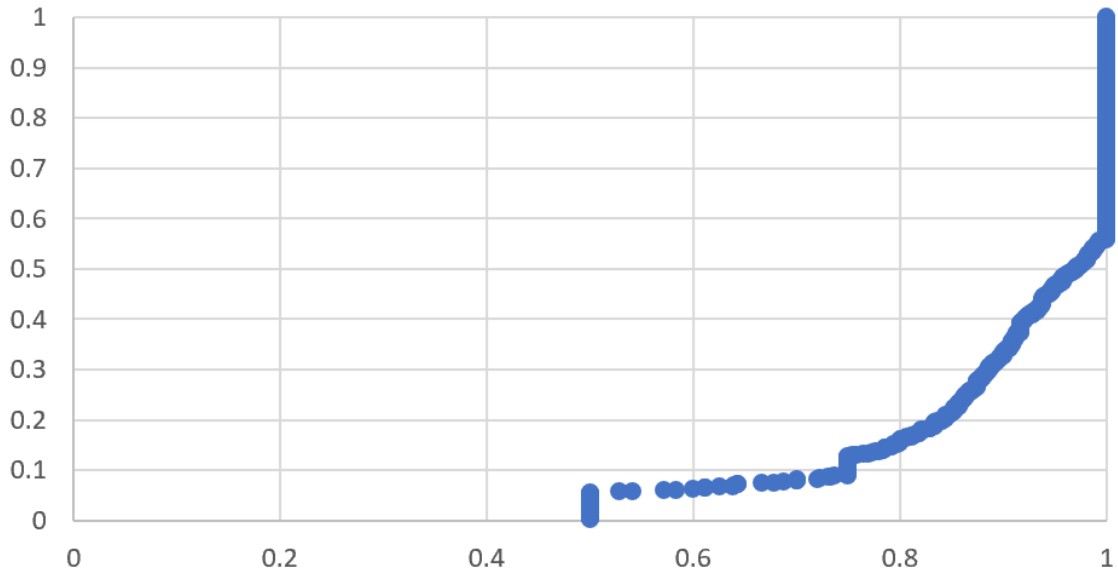


Figure 4.6: Distribution of flip rate values for tests

Opting for a threshold of 0.8 would lead to flagging 16% of the tests for further investigation, amounting to 110 tests out of the total 695 in this dataset. It’s worth noting that, as previously mentioned in 4.1, we initially had 970 unique tests. However, 276 of them were excluded from the analysis of flip rates for individual versions since they lacked either pass or fail results.

Almost 65% of the tests have a flake rate of less than 0.2 and 66% of tests have been rerun only once after failure. The flip rate is not a helpful metric for evaluating the flakiness level of tests in the Chromium project due to the overall invariability of its values.

4.6.2 RQ2: Can we discern legitimate failures from flaky tests?

To answer this research question, we apply a variety of machine-learning models to our dataset to predict legitimate failures from flaky ones.

We trained our models on 1,264,973 flaky tests and 9,323 fault-revealing tests. Then, we evaluated it on 217,503 failures caused by flaky tests and 2,320 fault-triggering failures caused by fault-revealing tests.

Model A: Bag of Words with Random Forest. As we had success with our proposed model in predicting the lifetime of flaky tests, we started our investigation with the same model. We applied a

balanced random forest with CountVectorizer to classify the dataset into legitimate failures and flaky failures. Table 4.5 reports the obtained performance. Similar to the performance achieved by previous vocabulary-based models on other datasets, our model was able to reach high accuracy with a precision of 99.1% and a recall of 97.8%. However, a noteworthy concern arises from a substantial false-positive rate, attributed to 80.2% of fault-triggering failures being erroneously classified as flaky (FP). This misclassification is critical, particularly considering that fault-triggering failures inherently signify genuine faults. The overall Matthews Correlation Coefficient (MCC) value stands at 0.13, indicating a relatively modest performance. This suggests that the model encounters challenges, compared to random selection, in accurately identifying fault-triggering failures.

Table 4.5: Performance of Random Forest Model. When using only static features (source code) model missed 77.4% of legitimate failures. Contribution of execution features improves the model performance.

Feature	Precision	Recall	MCC	FPR
Test Source	99.3%	97.1%	0.10	77.4%
Test Source + Execution Features	99.3%	87.6%	0.13	52.4%

To further examine the performance of our model, we look into the confusion matrix depicted in Figure 4.7. The x-axis reports the predicted label and the y-axis the actual label. Correct classifications are displayed in the top left (TN) and bottom right (TP). We observe that the model can detect flaky failures with high precision. We also see that 1,228 flaky tests are classified as legitimate failure (FN). This number is also important to consider: it translates in all cases where developers will be required to investigate irrelevant failures.

It is important to consider the number of legitimate failures that are wrongly identified as flaky (FP). Previous studies by researchers [7] [20] [21] have reported high precision, recall, and F1-Score when predicting flaky tests. However, our findings suggest that although we achieved similar performance, the high rate of false positives indicates that the model is biased towards the majority class and is unable to accurately identify these samples.

Model B: Word2Vec with Weighted Logistic Regression.

Weighted LR is used in classification problems with unbalanced datasets to help the model

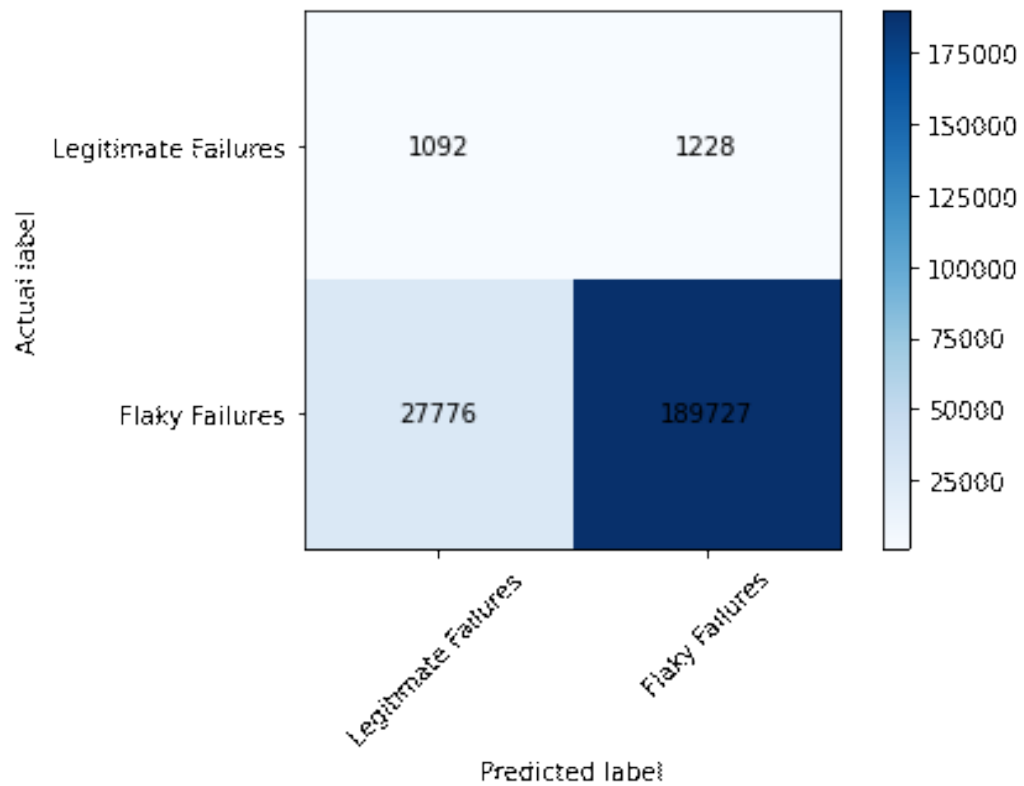


Figure 4.7: Confusion matrix for the model trained on source code and dynamic features. High accuracy is reached, similar to the performance reported in previous works. Nonetheless, 1,228 (52.9%) out of the 2,320 fault-triggering failures are mislabeled as flaky.

generate more balanced predictions. In our case, we started by giving a weight of 100 to the minority class and 1 to the majority class and increased this number to improve the results. Table 4.6 shows the result of this model.

Table 4.6: Model B Performance: Word2Vec with Weighted Logistic Regression

Precision	Recall	MCC	FPR
98%	98.9%	0.20	76.2%

Model C: BERT.

Table 4.7 represents the result of the Bert mode. Probably not very surprisingly, this model completely fails to capture legitimate failures. This poor performance can be related to only relying on the source code for prediction, the complexity of the model and the unbalanced nature of the dataset.

Table 4.7: Model C Performance: BERT

Precision	Recall	MCC	FPR
98.9%	100%	0	100%

The outcomes obtained from three distinct models, which fail to demonstrate promising results, lead us to believe that discerning fault-triggering flaky failures from general flaky failures resembles the search for a needle in a haystack.

Despite high precision and recall in predicting flaky failures, vocabulary-based models fail to capture fault-revealing failures with an MCC value remaining below 0.2.

4.6.3 RQ3: Are some test suites easier to predict than others?

In our effort to enhance the performance of the model proposed in [10], we experimented with a wide array of parameters. Our findings lead us to the conclusion that, given the highly unbalanced nature of the dataset, distinguishing legitimate failures from flaky ones becomes akin to locating a needle in a haystack. Different sampling techniques and the implementation of weighted classes

were attempted to address the issue of an imbalanced dataset, but these strategies proved ineffective.

Table 4.8: Percentage of Test Suit Categories in Dataset. Layout tests constitute 80% of the dataset, whereas unit tests account for a mere 8%. Across all three categories, legitimate failures are approximately one percent of the total tests.

Test Suit	Percentage Of Dataset	Legitimate Failures Percentage	Mean Flake Rate
Layout Tests	80.75%	1%	0.2
Browser Tests	10.7%	1%	0.2
Unit Tests	8.2%	0.7%	0.8

The inherent challenge lies in effectively distinguishing between the various categories within this imbalanced dataset. For all three categories, the ratio of legitimate failures is about 1% of all the tests, which makes the problem of discerning legitimate failures from flaky failures very challenging.

Table 4.8 outlines the distribution of test categories with Layout Tests comprising the majority of the dataset at 80.75%, followed by Browser Tests at 10.7% and Unit Tests at 8.2%. Across all categories, the occurrence of legitimate failures consistently hovers around one percent, indicating a uniform prevalence of legitimate failures within the test suits. Furthermore, the mean flake rates for Layout Tests and Browser Tests are recorded at 0.2, while Unit Tests exhibit a slightly higher flake rate of 0.8. The slightly higher flake rates observed for Unit Tests compared to Layout and Browser Tests imply that this category is more prone to flakiness. However, further investigation is needed to understand the factors contributing to this difference in flake rates among different test categories.

We use box plots to provide a more detailed analysis of the distribution of flake rates within test suite categories. Box plots are valuable tools for summarizing statistical information and comparing different groups of data. Figure 4.8 demonstrates the overall flake rate distribution of test suits. Notably, the box plots for Layout tests and Browser tests exhibit similar patterns, with approximately 50% of the tests showing a flake rate falling within the range of 0.1 to 0.5. Conversely, for Unit tests, this number ranges between 0.2 and 1, indicating a comparatively higher flake rate for the unit test category. Additionally, the median flake rate for Layout tests and Browser tests is observed to

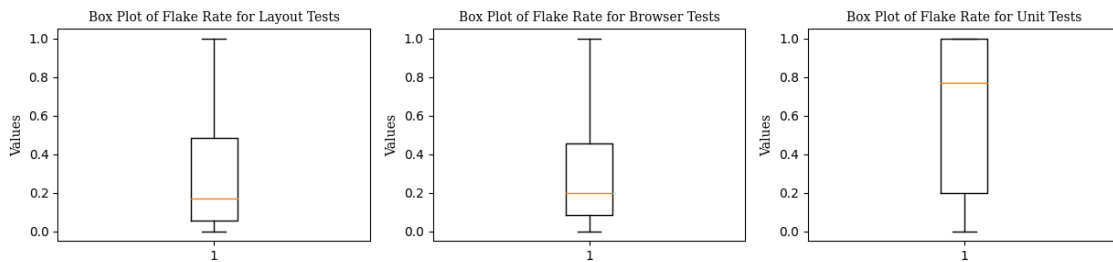


Figure 4.8: Box Plot of flake rate for different test suits. Unit tests have a higher flake rate with a mean of 0.8, while browser tests and layout tests have a mean of 0.2.

be 0.2, while for Unit tests, this measure stands at 0.8.

Figure 4.9 compares the distribution of flake rate for flaky failures and legitimate failures within each category.

We applied our models to these categories to see if the models could perform better within each category but perhaps not very surprisingly we reached the same result as the challenge of an unbalanced dataset still exists.

Layout tests embody 80% of the dataset and the ratio of flaky failures to legitimate failures stays the same as the whole dataset in subcategories. Unit tests have the highest flake rate among all the categories.

4.7 Threats to Validity

4.7.1 Internal Validity.

In this study, our objective was to leverage a dataset derived from an industrial project, specifically one generated through Continuous Integration (CI) builds. Our selection of the Chromium project was motivated by several factors, including its extensive test suite, open-source nature, and adherence to CI/CD practices. However, this decision necessitated certain trade-offs, notably the omission of certain features due to the project's scale. For instance, features related to test smells or test code coverage were absent from our dataset, which could have potentially enriched our predictive models for flaky failures. Despite these limitations, the Chromium dataset provided valuable

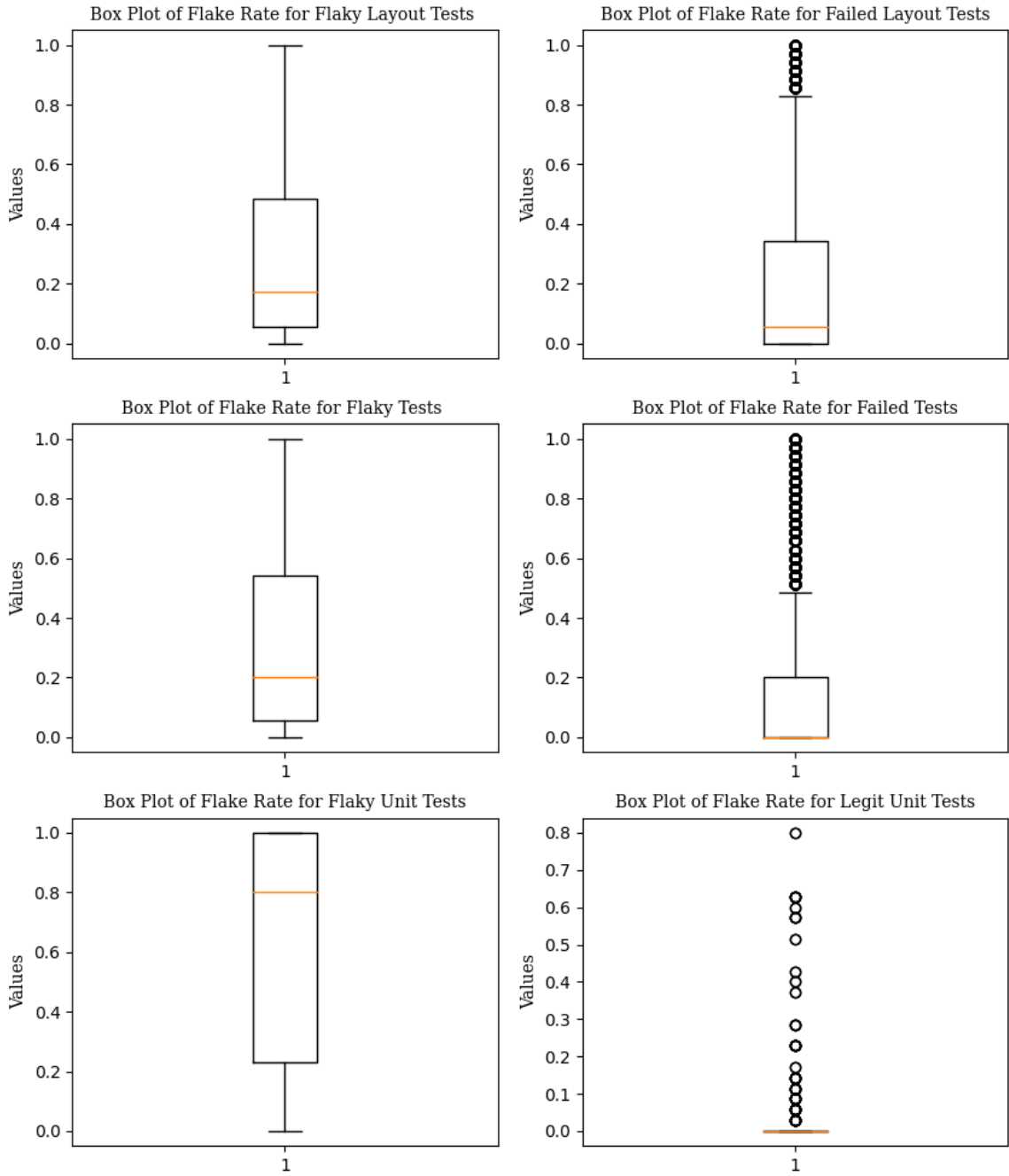


Figure 4.9: Box Plot of flake rate distribution for flaky failures vs. failed tests within each test suit.

insights into the prevalence and characteristics of flaky tests within a real-world software development environment. Going forward, it's important to explore potential ways to incorporate additional features and enhance the robustness of our predictive models.

4.7.2 External Validity.

In this study, our analysis focused on the dataset derived from the Chromium project. While Chromium serves as a valuable case study, it's essential to recognize that the generalizability of our findings to other software projects may be limited. Variations in project structures, development practices, and testing environments across different software systems could potentially influence the performance of the models discussed in our study. Additionally, it's important to note that our dataset predominantly comprises web/GUI tests, which inherently differ in their nature and characteristics from tests written in other programming languages. The unique intricacies and challenges associated with HTML and JavaScript testing may not directly translate to testing scenarios in different programming paradigms. Therefore, while our study provides insights into flakiness within the context of web/GUI testing, further research is needed to explore how these findings apply to diverse software projects and testing contexts.

4.7.3 Construct Validity.

A notable concern regarding the robustness of our study lies in the definition and identification of legitimate failures. The dataset utilized, as proposed by [10], relies on reruns as a means to classify failures. It is assumed that all fault-revealing tests within our dataset accurately pinpoint one or multiple issues within the codebase. However, it's crucial to acknowledge that some fault-revealing tests may exhibit flaky behavior due to factors such as insufficient execution time or the limited number of reruns. Consequently, certain tests categorized as fault-revealing may actually be flaky tests if they were subjected to additional reruns. Although efforts were made in the original study to exclude builds that were failing consecutively, this concern regarding misclassification remains valid.

4.8 Conclusion

In this chapter, we investigated the performance of existing vocabulary-based flaky test prediction methods in an industrial setting. We first used a dataset of 27 million tests with 21,932 flaky failures over a period of 30 days to investigate the magnitude of flakiness in the chromium dataset. We started our study by checking how often tests failed without any underlying issues within the code. Out of the 99,723 unique tests we analyzed, we found that 970 of them failed inconsistently, making up around 1% of all tests. Upon detailed examination of individual tests, it became evident that over half of them displayed failure rates of less than 9%, indicating sporadic instances of flakiness. However, a subset of tests demonstrated failure rates exceeding 70%, suggesting the presence of more significant underlying issues.

Furthermore, our investigation into the temporal variability of test results uncovered insights into the re-run behavior and flip rates of flaky tests. It is noteworthy that the majority of tests underwent only a single rerun, highlighting the common practice of not repeating tests. Additionally, our analysis of the frequency of transitions in test outcomes from failure to success or vice versa revealed that 86% of tests consistently demonstrated such transitions. However, the adoption of this transition rate as a reliable metric for ranking tests may be questionable, as a majority of tests exhibited similar rates of transition.

In the second part of this chapter, we used a more detailed dataset of 23,374 flaky tests and 2,343 fault-revealing (legitimate failures) tests over a period of 9 months. Employing different machine learning models, we achieved similar performance compared to previous studies in terms of precision and recall. However, a notable concern emerged regarding the substantial false-positive rate, highlighting the challenge of accurately identifying fault-triggering failures. Despite attempts to improve model performance through parameter optimization and sampling techniques, the inherent imbalance in the dataset posed significant challenges. It is important to consider the number of legitimate failures that are wrongly identified as flaky (FP). Previous vocabulary-based studies have reported high precision, recall, and F1-Score when predicting flaky tests. However, our findings suggest that although we achieved similar performance, the high rate of false positives indicates that the model is biased towards the majority class and cannot accurately identify these samples.

We also suggest that the source code of tests does not contain enough predictability power and other dynamic features like runtime duration need to be integrated into the model for optimal results. Overall, our findings emphasize the complexity of distinguishing legitimate failures from flaky ones, underscoring the need for further research and innovative approaches in this domain.

Looking forward, the findings of this study illuminate several avenues for future research and advancement in the field of flaky test prediction. We suggest that researchers prioritize the seamless integration of flaky test prediction mechanisms into established continuous integration and delivery (CI/CD) pipelines for more reliable results. One pivotal area of focus lies in the exploration of novel feature engineering techniques and model refinement strategies to enhance the predictive efficacy of machine learning algorithms. By delving beyond the surface-level analysis of source code vocabulary and incorporating more dynamic factors such as test dependencies, researchers can uncover intricate patterns and correlations within the dataset. Addressing the challenge of class imbalance through sophisticated sampling methods and ensemble learning approaches is another imperative task for future investigations. By embracing these directions, researchers can contribute to the development of more robust solutions, ultimately enhancing the reliability and quality of predictive models.

Chapter 5

Conclusion

The non-deterministic nature of flaky tests, characterized by unpredictable passing or failing outcomes, instills a sense of mistrust in the reliability of test automation. Such unpredictability poses a potential threat to the software development processes of numerous companies heavily dependent on automated tests for sustaining a continuous integration and delivery environment. Consequently, both industry practitioners and researchers are actively engaged in exploring strategies for preventing and identifying potential flaky tests, recognizing the critical need to fortify the robustness of automated testing frameworks in the realm of software development.

While software testing has long acknowledged flakiness as a prominent issue, research on this subject has rarely focused on classifying flaky tests for investigation in a large-scale industrial CI setting. In this research, we collected the history of more than 8K flaky tests on the Chrome project to be utilized for our lifetime classification model. We observe that 40% of the flaky tests remain unresolved. Among the flaky tests that undergo resolution or removal, 33% are addressed within a timeframe of fewer than 10 days. Identifying the appropriate tests to address is crucial, given that we aim to avoid allocating developers' time to the 40% of tests that may prove unsolvable. Our preference lies in pinpointing tests that are both more manageable and hold greater importance for timely resolution.

To effectively prioritize tests for investigation, it is crucial to analyze the distribution of the time it takes for a flaky test in Chrome to be resolved. This exploration is fundamental for establishing a robust framework to address the underlying issues. By analyzing the distribution of fixing time of

flaky tests we observe that except for two brief spikes observed over a 3-year period, our analysis indicates that the overall count of flaky tests remains relatively stable. This trend demonstrates that while developers address flaky tests, new ones continue to arise, maintaining a relatively consistent count of existing flaky tests. This shows the necessity for an efficient test prioritization method to direct developers' efforts towards reducing this trend.

We consider the lifetime of a test as a measure of the effort needed to address the test and its priority for being investigated. By solely analyzing the test description, our model can accurately predict the lifespan of flaky tests, achieving a precision of 73% and an MCC of 0.39. Analyzing test descriptions alone yields a high-precision method for predicting flaky test lifespans, suggesting a practical approach to test management.

To mitigate the effects of flakiness, both researchers and industrial experts proposed strategies and tools to detect and isolate flaky tests. To aid with these tasks, we applied state-of-the-art flakiness prediction methods at the Chromium CI and checked their performance. Perhaps surprisingly, we find that the application of such methods led to numerous faults missed, which is approximately 3/4 of all regression faults. In the second part of this thesis, we studied the behavior of flaky tests in a large-scale industrial setting. We used the results of Chromium CI builds to analyze the prevalence and scale of flaky tests.

We started our investigation by evaluating the effectiveness of metrics such as flake rate and flip rate in discerning different levels of flakiness. The result of our analysis showed that these measures are ineffective in ranking flaky tests in the case of the Chromium project. However, they provide insights into the flakiness scale of the Chromium project.

Subsequently, we directed our research efforts towards utilizing vocabulary-based approaches to distinguish flaky failures from faults that trigger failures. We used a dataset of 1.8 million test failures, representing the actual development process of more than 10,000 builds spanning a period of 9 months.

We examined the effectiveness of prior studies in predicting and ranking flaky tests within a large-scale CI/CD environment, specifically emphasizing vocabulary-based methods. To achieve this objective, we assessed the performance of the leading vocabulary-based flaky test prediction methods using 23,374 flaky tests and 2,343 fault-revealing (legitimate failures) tests.

Our study results indicate that vocabulary-based methods are unable to identify fault-revealing failures due to their sole reliance on source code and their lack of application in a large-scale industrial project within a CI setting. Additionally, we underscored the importance of prioritizing the prediction of flaky failures over flaky tests.

Bibliography

- [1] J. Micco. Flaky tests at google and how we mitigate them. [Online]. Available: <https://testing.googleblog.com/2016/05/flaky-tests-at-google-and-how-we.html>
- [2] O. Parry, “A survey of flaky tests,” *ACM Transactions on Software Engineering and Methodology*, vol. 31, no. 1, 2021.
- [3] W. Lam, P. Godefroid, S. Nath, A. Santhiar, and S. Thummalapenta, “Root causing flaky tests in a large-scale industrial setting,” in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2019, pp. 101–111.
- [4] L. Qingzhou, F. Hariri, L. Eloussi, and D. Marinov, “An empirical analysis of flaky tests,” in *Proceedings of the SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, 2014, pp. 643–653.
- [5] A. Romano, Z. Song, S. Grandhi, W. Yang, and W. Wang, “An empirical analysis of ui-based flaky tests,” in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. Madrid, Spain: IEEE, 2021, pp. 1585–1597.
- [6] W. Lam, R. Oei, A. Shi, D. Marinov, and T. Xie, “idflakies: A framework for detecting and partially classifying flaky tests,” in *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*. Los Alamitos, CA, USA: IEEE Computer Society, 2019, pp. 312–322. [Online]. Available: <https://doi.ieeecomputersociety.org/10.1109/ICST.2019.00038>
- [7] G. Pinto, B. Miranda, S. Dissanayake, M. d’Amorim, C. Treude, and A. Bertolino, “What is the vocabulary of flaky tests?” in *Proceedings of the 17th International Conference on Mining*

- Software Repositories*, ser. MSR '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 492–502. [Online]. Available: <https://doi.org/10.1145/3379597.3387482>
- [8] T. M. King, D. Santiago, J. Phillips, and P. J. Clarke, “Towards a bayesian network model for predicting flaky automated tests,” in *IEEE the International Conference on Software Quality, Reliability and Security Companion*. IEEE, 2018, pp. 100–107.
- [9] K. N. Emily Kowalczyk and Z. Gao, “Modeling and ranking flaky tests at apple,” (*ICSE SEIP*), 2020.
- [10] G. Haben, S. Habchi, M. Papadakis, M. Cordy, and Y. Le Traon, “The importance of discerning flaky from fault-triggering test failures: A case study on the chromium ci,” *arXiv preprint arXiv:2302.10594*, 2023.
- [11] J. Bell, O. Legunsen, M. Hilton, L. Eloussi, T. Yung, and D. Marinov, “Deflaker: Automatically detecting flaky tests,” in *Proceedings of the 40th International Conference on Software Engineering*, ser. ICSE '18. New York, NY, USA: ACM, 2018, pp. 433–444. [Online]. Available: <http://doi.acm.org/10.1145/3180155.3180164>
- [12] M. Machalica, A. Samylkin, M. Porth, and S. Chandra, “Predictive test selection,” in *Proceedings of the 41st International Conference on Software Engineering: Software Engineering in Practice*, ser. ICSE-SEIP '19. IEEE Press, 2019, p. 91–100. [Online]. Available: <https://doi.org/10.1109/ICSE-SEIP.2019.00018>
- [13] A. Vahabzadeh, A. M. Fard, and A. Mesbah, “An empirical study of bugs in test code,” in *2015 IEEE international conference on software maintenance and evolution (ICSME)*, 2015, pp. 101–110.
- [14] S. Thorve, C. Sreshtha, and N. Meng, “An empirical study of flaky tests in android apps,” in *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2018, pp. 534–538.
- [15] F. Palomba and A. Zaidman, “Does refactoring of test smells induce fixing flaky tests?” (*IC-SME*), 2017.

- [16] M. H. U. Rehman and P. C. Rigby, “Quantifying no-fault-found test failures to prioritize inspection of flaky tests at ericsson,” in *Proceedings of the 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, 2021.
- [17] H. Spieker, A. Gotlieb, D. Marijan, and M. Mossige, “Reinforcement learning for automatic test case prioritization and selection in continuous integration,” in *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*., 2017, pp. 12–22.
- [18] H. Jahan, Z. Feng, S. Mahmud, and P. Dong, “Version specific test case prioritization approach based on artificial neural network,” *Journal of Intelligent and Fuzzy Systems*, pp. 6181–6194, 2019.
- [19] S. Yoo, M. Harman, P. Tonella, and A. Susi, “Clustering test cases to achieve effective and scalable prioritisation incorporating expert knowledge,” in *Proceedings of the eighteenth international symposium on Software testing and analysis*., 2009, pp. 201–212.
- [20] B. H. P. Camara, M. A. G. Silva, A. T. Endo, and S. R. Vergilio, “What is the vocabulary of flaky tests? an extended replication,” in *2021 IEEE/ACM 29th International Conference on Program Comprehension (ICPC)*. IEEE, 2021, pp. 444–454.
- [21] S. Fatima, T. A. Ghaleb, and L. Briand, “Flakify: A black-box, language model-based predictor for flaky tests,” *IEEE Transactions on Software Engineering*, 2022.
- [22] G. Haben, S. Habchi, M. Papadakis, M. Cordy, and Y. Le Traon, “A replication study on the usability of code vocabulary in predicting flaky tests,” in *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*. IEEE, 2021, pp. 219–229.
- [23] H. Aman, S. Amasaki, T. Yokogawa, and M. Kawahara, “A comparative study of vectorization-based static test case prioritization methods,” in *2020 46th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*. IEEE, 2020, pp. 80–88.

- [24] R. Pan, M. Bagherzadeh, T. A. Ghaleb, and L. Briand, “Test case selection and prioritization using machine learning: a systematic literature review,” *Empirical Software Engineering*, vol. 27, no. 2, p. 29, 2022.
- [25] A. Bertolino, E. Cruciani, B. Miranda, and R. Verdecchia, “Know your neighbor: Fast static prediction of test flakiness,” in *Proceedings of the International Conference on Software Engineering (ICSE)*, 2020, istituto. [Online]. Available: <https://ieeexplore.ieee.org>
- [26] B. Camara, M. Silva, A. Endo, and S. Vergilio, “On the use of test smells for prediction of flaky tests,” in *Brazilian Symposium on Systematic and Automated Software Testing*, 2021, pp. 46–54, cit. on pp. 74, 77, 80, 85, 87, 98, 103, 120, 130.
- [27] A. Alshammari, C. Morris, M. Hilton, and J. Bell, “Flakeflagger: Predicting flakiness without rerunning tests,” in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, 2021, pp. 1572–1584, (cit. on pp. 26, 27, 30, 77, 80, 81, 98, 115, 116, 120, 129, 130).
- [28] V. Pontillo, F. Palomba, and F. Ferrucci, “Toward static test flakiness prediction: A feasibility study,” in *Proceedings of the 5th International Workshop on Machine Learning Techniques for Software Quality Evolution*, 2021, pp. 19–24, (cit. on p. 26).
- [29] Y. Qin, S. Wang, K. Liu, and et al., “Peeler: Learning to effectively predict flakiness without running tests,” in *2022 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. Limassol, Cyprus: ICSME, 2022, pp. 257–268, (cit. on pp. 26, 27).
- [30] D. Olewicki, M. Nayrolles, and B. Adams, “Towards language-independent brown build detection,” in *Proceedings of the 44th International Conference on Software Engineering*, ser. ICSE ’22. Association for Computing Machinery, 2022, pp. 2177–2188, (cit. on pp. 26, 120, 131). [Online]. Available: <https://doi.org/10.1145/3510003.3510122>
- [31] A. Akli, G. Haben, S. Habchi, M. Papadakis, and Y. Le Traon, “Predicting flaky tests categories using few-shot learning,” in *Proceedings of the 34th ACM/IEEE International Conference on Automation of Software Test*, ser. AST ’23. Melbourne, Australia: Association for Computing Machinery, 2023, (cit. on pp. 26, 73, iii).

- [32] Neighbor, “Fast static prediction of test flakiness,” in *Proceedings of the International Conference on Software Engineering (ICSE)*, 2020, (cit. on pp. 26, 49, 71, 120, 131). [Online]. Available: <https://ieeexplore.ieee.org>
- [33] S. Malmir and P. C. Rigby, “Predicting the lifetime of flaky tests on chrome,” in *2024 International Flaky Tests Workshop (FTW '24)*. Lisbon, Portugal: ACM, April 14 2024, p. 9.
- [34] P. M. Duvall, S. Matyas, and A. Glover, *Continuous Integration: Improving Software Quality and Reducing Risk*. Pearson Education, 2007.
- [35] Flake portal. [Online]. Available: https://chromium.googlesource.com/infra/infra/+master/appengine/findit/docs/flake_portal.md#flake-portal
- [36] Web test expectations and baselines. [Online]. Available: https://chromium.googlesource.com/chromium/src/+master/docs/testing/web_test_expectations.md
- [37] M. Eck, F. Palomba, M. Castelluccio, and A. Bacchelli, “Understanding flaky tests: The developer’s perspective,” in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering.*, 2019, pp. 830–840.
- [38] E. Fallahzadeh and P. C. Rigby, “The impact of flaky tests on historical test prioritization on chrome,” in *Proceedings of the 44th International Conference on Software Engineering: Software Engineering in Practice*, 2022.
- [39] S. learn developers. (2022) Sklearn.feature_selection.selectkbest — scikit-learn 1.1.2 documentation. (Accessed on 08/11/2022), Aug. 2022 (cit. on p. 131). [Online]. Available: https://scikit-learn.org/stable/modules/generated/sklearn.feature_selection.SelectKBest.html
- [40] J. Micco. (2016, May 27) Flaky tests at google and how we mitigate them. [Online]. Available: <https://testing.googleblog.com/2016/05/flaky-tests-at-google-and-how-we.html>
- [41] M. Rehkopf. What is continuous integration — atlassian. (cit. on pp. 54, 74, 120). Accessed on 01/12/2021. [Online]. Available: <https://www.atlassian.com/continuous-delivery/continuous-integration>

- [42] C. Leong, A. Singh, M. Papadakis, Y. L. Traon, and J. Micco, “Assessing transition-based test selection algorithms at google,” in *Proceedings of the 41st International Conference on Software Engineering: Software Engineering in Practice, ICSE (SEIP) 2019*, H. Sharp and M. Whalen, Eds. Montreal, QC, Canada: IEEE / ACM, May 25-31 2019, pp. 101–110, (cit. on pp. 11, 24, 54, 74, 98, 120, 126). [Online]. Available: <https://doi.org/10.1109/ICSE-SEIP.2019.00019>
- [43] T. Durieux, C. Le Goues, M. Hilton, and R. Abreu, “Empirical study of restarted and flaky builds on travis ci,” in *Proceedings of the 17th International Conference on Mining Software Repositories*, 2020, pp. 254–264, (cit. on p. 24).
- [44] C. Ziftei and D. Cavalcanti, “De-flake your tests: Automatically locating root causes of flaky tests in code at google,” in *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2020, pp. 736–745.
- [45] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean, “Distributed representations of words and phrases and their compositionality,” in *Advances in neural information processing systems 26*, 2013.
- [46] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, “Bert: Pre-training of deep bidirectional transformers for language understanding,” *arXiv preprint*, 2018.
- [47] Chromium layout tests documentation. (Accessed on May 3, 2024). [Online]. Available: https://chromium.googlesource.com/chromium/src/+61.0.3158.0/docs/testing/layout_tests.md
- [48] C. Project. (2023) Running tests - chromium development documentation. (Accessed on May 3, 2024). [Online]. Available: https://www.chromium.org/developers/testing/running-tests/#:~:text=Unit%20tests%20verify%20some%20part,with%20a%20_browser%20test.cc%20suffix
- [49] ——. (2023) Browser tests - chromium development documentation. (Accessed on May 3, 2024). [Online]. Available: <https://www.chromium.org/developers/testing/browser-tests/>

[50] Unit testing - google chrome extensions. (Accessed on May 3, 2024). [Online]. Available: <https://developer.chrome.com/docs/extensions/how-to/test/unit-testing#:~:text=Unit%20testing%20allows%20small%20sections,writes%20a%20value%20to%20storage>