FlaKat: A Machine Learning-Based Categorization Framework for Flaky Tests

by

Shizhe Lin

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Applied Science
in
Computer Engineering

Waterloo, Ontario, Canada, 2023

Author's Declaration

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

Flaky tests can pass or fail non-deterministically, without alterations to a software system. Such tests are frequently encountered by developers and hinder the credibility of test suites. Thus, flaky tests have caught the attention of researchers in recent years. Numerous approaches have been published on defining, locating, and categorizing flaky tests, along with auto-repairing strategies for specific types of flakiness. Practitioners have developed several techniques to detect flaky tests automatically. The most traditional approaches adopt repeated execution of test suites accompanied by techniques such as shuffled execution order, and random distortion of environment. State-of-the-art research also incorporates machine learning solutions into flaky test detection and achieves reasonably good accuracy. Moreover, strategies for repairing flaky tests have also been published for specific flaky test categories and the process has been automated as well. However, there is a research gap between flaky test detection and category-specific flakiness repair.

To address the aforementioned gap, this thesis proposes a novel categorization framework, called FlaKat, which uses machine-learning classifiers for fast and accurate categorization of a given flaky test case. FlaKat first parses and converts raw flaky tests into vector embeddings. The dimensionality of embeddings is reduced and then used for training machine learning classifiers. Sampling techniques are applied to address the imbalance between flaky test categories in the dataset.

The evaluation of FlaKat was conducted to determine its performance with different combinations of configurations using known flaky tests from 108 open-source Java projects. Notably, Implementation-Dependent and Order-Dependent flaky tests, which represent almost 75% of the total dataset, achieved F_1 scores (harmonic mean of precision and recall) of 0.94 and 0.90 respectively while the overall macro average (no weight difference between categories) is at 0.67.

This research work also proposes a new evaluation metric, called Flakiness Detection Capacity (FDC), for measuring the accuracy of classifiers from the perspective of information theory and provides proof for its effectiveness. The final obtained results for FDC also aligns with F_1 score regarding which classifier yields the best flakiness classification.

Acknowledgements

I would like to first thank my supervisor Dr. Ladan Tahvildari. This thesis will not exist without her help and guidance along the way. Her endorsement and encouragement motivates me to take the path of research and explore my life from a different perspective. It is my pleasure to work with her and I am looking forward to upcoming collaborations.

I learnt many valuable lessons from Dr. Mark Crowley for data modeling and Dr. Patrick Lam for static analysis. They also spend great efforts providing feedback for my thesis which I am sincerely grateful. I also want to thank the members of the STAR group, especially Ryan Zheng He Liu, for the discussion and assistance at the early stage of the thesis development.

I would also like to thank my parents, Lin Gesheng and Pan Weiping, for their never-changing support throughout the years. The pandemic makes it difficult to meet in person but I will always appreciate the experience and wisdom they shared through screen.

Finally, I would like to thank my friends in UW Kendo Club and Animusic Band. Both the sword fight and music performance are new and fun adventures for me. They kept my morale high and I enjoy the flavor they brought to my research life.

Dedication

To the wonderful people helping and supporting me throughout this journey.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

Continuous Integration and regression testing are widely adopted in both industry and open-source community for software development and release [32, 77]. Test cases are written to verify that production code functions as expected and buggy test cases are not that uncommon [42]. A fault in test code may cause the test to miss a bug in the production code [86] but the ones that fail while the production code is correct also important. Those tests are called *flaky* tests [43] which can pass or fail non-deterministically in different executions, without altering the software under test, which undermines the credibility of regression tests. They have been widely studied in the community and numerous works are published on defining [55, 68], locating [8, 29, 46], and categorizing them along with fixing strategies for the specific type of flakiness [68, 80, 103]. The traditional solutions for flaky test detection are extremely expensive in terms of computational power and runtime which led to the development of machine learning-based flaky test detection tools [4, 71, 25]. However, the state-of-the-art machine learning approaches solely focus on predicting whether a test case is flaky or not.

Motivated by the recent work in incorporating machine learning algorithms into the domain of flaky test detection, this thesis proposes a novel framework, FlaKat. It takes one step further and adopts machine learning classifiers for fast and accurate predictions of the *category* of known flaky tests. The framework fills the existing research gap between the numerous flaky test detection tools and category-specific flaky test repair solutions. Various source code vectorization techniques and other components in the framework are analyzed and evaluated using known flaky tests from 108 open-source Java projects. To the best knowledge of the author, the proposed framework is the first static flaky test *categorization* tool that uses machine learning.

The evaluation of FlaKat is firstly conducted in F_1 score and shows promising performance. Additionally, this thesis also proposes Flakiness Detection Capacity (FDC) as a new metric to measure the accuracy of the machine learning classifiers in predicting the category of flaky tests. It is inspired by a previous work in intrusion detection [28] and quantifies the performance from an information theory point of view. The value of FDC is proved and then applied to FlaKat for better comparison between the choice of components.

1.1 Research Challenge

The recent works that incorporate machine learning algorithms into flaky test detection have convincing results that the vector representation of flaky and non-flaky tests are different [4, 25, 71, 96]. They provide fast and accurate ways of detecting flakiness without spending time and resources on re-running test suites. However, there are some problems that need to be answered for machine learning flaky test categorization to be feasible.

Whether different categories of the flaky test defined in the current literature [29, 46, 48, 80] shows clustering in the vector space remains unknown. If source code vectorization methods yield mixed clusters with data points from different categories overlapping with each other, then it would be impossible for any classifiers to provide promising results.

Due to the high number of features in the vector embedding, a dimensionality reduction technique that preserves both local and global structures of the flaky test embedding is mandatory. It can help to reduce the training time of the machine learning classifiers to a reasonable scale without losing too much information. Reduced 2D projection also helps to provides answers from a visual perspective.

It is crucial to select machine learning classifiers such that they can accurately contribute in the categorization process. The result will be measured in F_1 score and both macro average and individual score will be examined. The weight distribution between the categories is undecided and that gives room for applying FDC.

Comparison between FDC and F_1 score is made from the perspective of consistency and discriminancy. Only when FDC shows superiority in both fields, then it can better measure the performance of machine learning classifiers in handling flaky test categorization.

1.2 Thesis Contribution

The FlaKat framework is designed and developed that aims to fill the gap between flaky test categorization [29, 46, 48, 80] and flaky test repairs of specific types [68, 80, 103] via a machine learning approach. First, the raw test cases will be vectorized into embeddings using the source code vectorization method. Then, several dimensionality reduction techniques will be applied to the embeddings which are later used for training and testing the machine learning models. Additionally, Flakiness Detection Capacity is implemented and proved to be valuable after comparison against F_1 score. Thus, the research contributions of the thesis to the research community are as follows:

The process of evaluating FlaKat shows that the different categories of flaky test cases maintained their difference after being converted into vector embeddings using doc2vec [50], code2vec [3] and td-idf [40]. The quality of data provided to the machine learning models has a crucial impact on their performance. After applying code vectorization, raw Java test cases are converted into arrays of floating points in vector space. If data points from the same flaky category do not form clusters and separated from data points from other categories, then accurate predictions become impossible. Reducing the dimensionality of these arrays to two helps to visualize the data points and the two-dimensional scatter plots of vectorized flaky tests indicate clustering in some reduced embedding using specific techniques and configurations.

The strength of several dimensionality reduction techniques, PCA [38], LDA [89], Isomap [87], t-SNE [94] and UMAP [59], selected in FlaKat, is able to preserve both local and global structures of the vector embedding of flaky tests. Different dimension reduction methods yield distinct results and strongly affect the performance of predictions later. Several popular reduction algorithms are applied and the optimal one will be selected. To evaluate dimensionality reduction algorithms used in FlaKat, direct inspection of reduced vector embedding is not accurate enough to draw a sound conclusion about which dimensionality reduction technique is most suitable for flaky test categorization. Based on the method in an existing work [59], applying KNN algorithm will be added more confidence to the selection of optimal technique and configuration. The results with a small k value represent the preservation of the local structure and results with a large k value represents the preservation of the global structure.

FlaKat achieves decent accuracy in predicting the category measured in F_1 score. With the embeddings reduced by Linear Discriminant Analysis (LDA), the macro average with all categories sharing equal weights achieved 0.64 when using SVM [17] and 0.67 when using Random Forest classifier [33]. F_1 score is calculated from the harmonic

mean of both precision and recall making it a widely-used metric for evaluating machine learning models. It provides a quantitative value on how well the classifier predicts with data points from the test set. The hyperparameters of different classifiers are also tuned and their performance is evaluated and compared.

This thesis introduces the novel metric of Flakiness Detection Capacity (FDC) and demonstrates its superiority compared to the F1 score, in terms of consistency and ability to discriminate. FDC shown in Equ.4.1 is inspired by earlier work in intrusion detection [28] which focuses on the correlation between the input and output of classifiers and can evaluate their performance from a different point of view. Its value is mathematically proved by calculating its relative consistency and discriminancy against F_1 score before applying it to the comparison of classifiers.

1.3 Thesis Organization

The rest of this thesis is organized into 4 chapters as follows:

Chapter 2 provides the background knowledge of this work. This chapter reviews the current literature and display the state-of-the-art of research regarding flaky tests. It considers different aspects of flaky tests including their definition, impact and detection. In particular, the existing machine learning flakiness detection frameworks are examined closely. The motivation behind the proposed framework FlaKat is also explained.

Chapter 3 illustrates the goals and the architecture of the proposed FlaKat framework. It explains the overall design of the framework and justifies the strategy to tackle the problems in the field of flaky tests. This chapter shows the different phases of the framework and their corresponding processes.

Chapter 4 shows the implementation details of the various components used in the realization of the framework from parsing of the raw data to making the final predictions. The required knowledge for each plug-and-play components used in FlaKat framework is also discussed in this chapter.

Chapter 5 presents the design of the experiments for evaluating combinations of selected options within the framework. This chapter provides the performance data and the analysis to fill the research gaps in the field. The value of the newly proposed Flakiness Detection Capacity is proved with respect to F_1 score and used in the evaluation of FlaKat as well.

Chapter 6 summarize the work accomplished in this thesis and conclude the previously mentioned topics. Threats to the validity are listed and several directions for future work are pointed out.

Chapter 2

Background and Related Work

This chapter presents the earlier research in the community and current state-of-the-art solutions. Section 2.1 explains the impact of flaky tests in different aspects of software testing. Section 2.2 lists various causes of flakiness and illustrates the current definitions of different categories of flaky tests. Section 2.3 elaborates on the existing frameworks on the detection of flaky tests and Section 2.4 presents how flaky tests are repaired and mitigated.

2.1 Definition and Impact of Flaky Test

Faulty code segments in a commit will cause regression tests to fail which informs software developers about the existence of bugs in their code. However, certain tests can fail in different executions without modifications to the software under test. Such tests are called flaky since the earliest reference in [43] because their outcome is non-deterministic with respect to a given software version.

Several aspects of testing are affected by flaky tests. First, the developers usually assume the outcome of tests is consistent and trustworthy with the same software under test but the existence of flaky tests causes the results to be non-deterministic and significantly hinders the credibility of the entire test suite [68] and developers could mistakenly spend unnecessary time debugging a potentially false alarm that is unrelated to any recent changes. Multiple studies have examined the manifestation of flaky tests in terms of indicating the presence of non-existent bugs, or conversely, missing real bugs [93, 104]. Second, it is often challenging and time-consuming to reproduce failures caused by flakiness due to their nature of non-determinism. One study empirically evaluated the reproducibility

of flaky tests on a large scale [24, 48] and sometime it would take up to 2000 runs for the flakiness to be detected [4]. Last, the false alarms created by flaky tests are not meaningless because their existence indicates underlying issues in the software. It is a common practice in the industry for developers to disable the known flaky tests in the test suite but such action may potentially leave the genuine bugs undetected. One source found that ignoring flaky test failures lead to a decrease on software stability, measured in the number of crash reports received for the Firefox web browser [72].

Due to their negative effects on the reliability of test results, researchers in the industry and academics have put numerous effort into the domain of the flaky test. A general investigation into the prevalence and nature of bugs in test code was performed by Vahabzadeh et al. [93]. Through their manual analysis, they identified 5,556 unique bug reports across projects of the Apache Software Foundation, 443 of which they systematically categorized. They categorized these by manifestation, that is, whether the test bug led to a false alarm or to a missed alarm and by their root cause. False alarm test bugs, while potentially less serious than "silent horrors" [93], can still take a considerable amount of time and effort for developers to debug. Since flaky tests are likely the dominant root cause of such bugs this means that they are potentially a significant waste of a developer's time. They found that test flakiness is responsible for 53% of the false alarm cases which is 97% of test bugs manifested in total. A similar conclusion is drawn in other works as well. Memon et al. [60] performed a study into reducing the cost of continuous testing at Google. They described flaky tests as a reality of practical testing in large organizations, citing them as a constraint to the deployment of the results of their work. Specifically, they identified approximately 0.8% of their total dataset of over five million build-ables and executable code units, as flaky. Of the 115,160 test targets that had historically passed at least once and failed at least once, flaky tests constituted 41%. Labuschagne et al. [42] studied the cost of regression testing in practice and they found that 18% of the total test suite executions fail. More interestingly, 13% of these failures are flaky.

Rahman et al [72] examined the impact of ignoring flaky test failures on the number of crash reports associated with builds of the Firefox web browser in both the beta and production release channels in their rapid release schedule [64]. To identify flaky tests, they searched for records of test executions marked with the phrase "RANDOM," a term commonly used by Firefox developers, in the testing logs. This technique is based on the find from Lou et al. [23] that keywords such as "intermittent" and "flak" within the commit history are likely to indicate test flakiness. In the testing logs of Firefox, they found that builds with one or more flaky test failures were associated with a median of 514 and 234 crash reports for the beta and production channels in the median case, respectively. Furthermore, a Wilcoxon signed-rank statistical test comparing the number

of crashes between the two channels showed a statistically significant difference. They concluded that developers were more conservative about releasing production builds with known flaky tests. For those with failing tests in general, these figures were 508 crash reports for the beta channel and 291 for the production channel. In the case of builds with all tests passing, they recorded a median of only two crash reports for each channel. These findings indicate that ignoring test failures, flaky or otherwise, appears to lead to a considerably higher volume of crashes due to missed bugs.

Microsoft's distributed build system, CloudBuild, was the object of analysis conducted by Lam et al. [44]. They identified 2,864 unique flaky tests across five projects by examining the logs from repeated test executions. Overall, of the 3,871 individual builds that they sampled, 26% presented flaky test failures. Furthermore, data from Flakes, the flaky test management system integrated into CloudBuild at Microsoft, was used in a later study on the lifecycle of flaky tests [45]. Over a 30-day period, they found that Flakes identified a total of 19,793 flaky test failures across six subject projects, representing just 0.02% of the sampled test executions which must be smaller compare to other studies in the industry. However, they also reported that flaky test failures would have been responsible for a total of 1,693 failed builds, which would have represented 5.7% of all the failed builds sampled. They concluded that the percentage of builds that may be affected by flaky test failures can be relatively significant even when flaky tests are not particularly prevalent.

When evaluating a data-driven test selection technique deployed at Facebook, Machalica et al. [57] remarked how flaky tests represented a significant obstacle to the accuracy of their approach. Their test selection technique used a classification algorithm, that was trained with previous examples of failed tests and their respective code changes, to only select tests for execution when it predicted that they might fail given a new code change. Finding that the set of flaky tests in their dataset was four times larger than the set of deterministically failing tests, they explained that if they did not filter flaky tests from their training data then they would run the risk of training their classifier to capture tests that failed flakily rather than those that failed deterministically due to a fault introduced by a code change. They also pointed out closely related work done by Memon et al. [60] where the observation is that changed units of code and failing tests have small distances in the build dependency graph and is use the distance as one of the features in their predictive test selection strategy.

In an ideal situation, test cases should be independent of each other which is not always true in practice, as reported in an empirical study [42]. Zhang et al. [104] performed an empirical study of test order dependencies by analyzing 96 real-world dependent tests from 5 issue tracking systems. They pointed out that test dependence can be difficult to identify by developers and described how such a phenomenon can mask bugs since the order in which

the containing test suite is executed might determine whether or not the dependent test is able to expose faults or not. They also explained how to test order dependencies might lead to spurious bug reports when the execution order of tests alters, potentially manifesting the order-dependent tests and exposing an issue in the test code rather than a bug in the code under test. Lam et al. [47] evaluated the consequences of order-dependent tests upon the soundness of regression test prioritization [37, 74, 75, 83], selection [9, 31, 34, 66, 67, 101] and parallelization [62]. They evaluated a variety of algorithms upon 11 modules of open-source Java projects with both developer-written and automatically generated test suites with the order-dependent tests already identified in previous work [46]. After applying test prioritization, they found a total of 23% of order-dependent tests in developer-written test suites failed and 54% of such tests in their automatically generated test suites failed, when they were previously passing. For test selection, this was 24% and 4% for the developer and automatic test suites, respectively, and, for parallelization, 5% and 36%, respectively. As part of an empirical study of test order dependence, Zhang et al. [104] assessed how many test cases gave inconsistent outcomes when applying five different test prioritization schemes across the test suites of four open-source Java projects. Their results showed that for one subject in particular with 18 previously identified order-dependent tests, up to 16 gave a different outcome compared to a non-prioritized test suite run. This finding indicated that the soundness of test prioritization was, at least for this project, significantly impacted by order-dependent tests—even though test prioritization methods are not supposed to affect the outcomes of tests at all. Many other techniques also assume test independence, including test execution [41], test factoring [76, 99], test carving [22], and experimental debugging techniques [84, 100, 102]. Thus the flakiness introduced by test order dependency could threat the validity of testing techniques previously mentioned.

Mutation testing is the practice of assessing the bug-finding capability of a test suite by generating many versions of a software under test with small syntactical perturbations, to resemble bugs, known as mutants [36]. A test is said to kill a mutant if it fails when executed upon it, thus witnessing the artificial bug. If a test covers mutated code but does not kill it, then the mutant is said to have survived. At the end of a test suite run, the percentage of killed mutants is calculated and is called a mutation score which will be used as a measurement of test suite quality. Demonstrating the extent to which flakiness can impact mutation testing, Shi et al. [78] conducted an experiment with 30 Java projects. In the context of this study, they focus their attention on the non-determinism observed in the coverage of tests rather than the actual test outcome. As an initial motivating study, they examined the number of statements that were inconsistently covered across their subjects when repeatedly executing their test suites. Overall, they found that the coverage of 22% of statements was non-deterministic. PIT [16], a mutation testing framework is used in

their work to generate mutants for these same subjects. The framework first runs the test suite to analyze the coverage of each test so that it generates mutants that the tests have a chance of killing. Depending on the level of flakiness in the coverage, they explained that when they are executed again to measure mutant killing, they may not even cover their mutants, such that their killed status is unknown. Investigating this phenomenon's extent by applying PIT to their subject set, they found that over 5% of mutants ended up with an unknown status due to inconsistent coverage, leading to uncertainty in the mutation score. Assuming that all unknown mutants were killed, the overall mutation score would have been 82% and—further assuming that they all survived—this score would have been 78%, showing that non-deterministic coverage can limit the reliability of this test effectiveness metric.

Vancsics et al. [95] studied the influence of flaky tests on automated fault localization techniques. Fault localization uses the outcomes and coverage of tests to identify the location of faulty program elements [84, 102] and current literature on program repair and fault localization typically does not consider the possible flakiness of testing environments. The performance of these approaches is predominantly investigated using bug benchmarks that rely on deterministic test case behaviour. They defined the flakiness-ratio metric to express the flakiness of a test, and conduct a simulation-based experiment using real projects with real faults. Results show that the different investigated algorithms are affected by flakiness at different levels. In general, Tarantula [39] is more sensitive to flakiness than Ochiai [2] and DStar [98]. However, results also indicate that this effect highly depends on the characteristics of the actual project, and the related test cases as well.

Najafi et al. [65] investigated the impact of flakiness on batch testing, a technique for improving efficiency within a continuous integration pipeline. Instead of testing each new commit individually, batch testing groups commits together to reduce test execution costs. Should the batch pass then each commit can proceed to the next stage of the pipeline. They modelled the degree of flakiness for each project and find that test flakiness reduces the cost savings to 42%. Should a batch fail, it has to be repeatedly bisected to identify the commits that caused the failure, effectively performing a binary search for the culprit. They explained that flaky tests are a threat to the applicability of batch testing, since a spurious flaky failure may lead to unnecessary bisections. To mitigate this, smaller batch sizes can be selected, since the flaky failure will affect fewer commits, though this limits the potential efficiency gains from applying batch testing in the first place. An evaluation of three unnamed projects at Ericsson demonstrated the following negative correlation: the more flakiness within a test suite, the smaller the most cost-effective batch size.

A later study by Paydar et al. [69] examined the prevalence of flaky tests within the regression test suites generated specifically by Randoop. They explained how regression

test suites are useful as far as they capture the behaviour of a program at a given point in time and are used to identify if a recent code change has any unintended effects. If a regression test suite contains flaky tests, then it does not accurately reflect the behaviour of the program and is thus less informative for developers. They took between 11 and 20 versions of five open-source Java projects and used Randoop to generate regression test suites, which were the main objects of analysis. Overall, they found that 5% of their automatically generated test classes were flaky, and on average, 54% of the test cases within each of these were flaky, further demonstrating that automatically generated tests can contribute to the flakiness of a test suite. Specifically, they employed a technique known as Random testing based on Ground Truth (RGT) to determine if a generated patch was correct or overfitting. A patch is considered overfitting if it passes the developer-written tests it was generated from, yet is generally a poor solution to the bug in question and may fail on tests held out from the patch generation process. To that end, the RGT technique uses an automatic test generation technique such as Randoop [69]. If a patch fails an automatically generated test, then it is labelled as overfitting. In this case, the test is also applied to a developer-written, ground truth patch to determine the behavioural difference with the overfitting patch. Given the potential for automatic test generation techniques to produce flaky tests [69], the RGT technique includes a preprocessing stage where it repeatedly executes each generated test on the ground truth patch. Given that the ground truth patch is considered to be correct, any test failure at this stage is considered to be flaky and the test is discarded. Having discarded 2.2% of tests generated by EvoSuite and 2.4% generated by Randoop for this reason, Ye et al. concluded that flaky test detection is an important consideration for researchers in automatic program repair. They remarked how this was a threat to the internal validity of their study, explaining how the flaky tests they discarded may still have exposed behavioural differences between generated and ground truth patches. As such, this could have led to them underestimating the effectiveness of RGT in the context of automated program repair.

Lam et al. [48] focus their attention on the non-order-dependent flaky tests [46]. They posited that when encountering a failing test during a test suite run, a developer is likely to run that test in isolation to reproduce the failure and thus debug the code under test. When trying to reproduce the inconsistent outcome of a flaky test, they demonstrated that this technique may not be effective. They executed the test suites of 26 modules of various open-source Java projects 4,000 times and calculated each test's failure rate—the ratio of failures to total runs. For each test they found to be flaky, meaning a failure rate greater than zero or less than one, they re-executed it 4,000 times again in isolation. They found that, of the 107 tests were identified as flaky, and 57 gave totally consistent outcomes in isolation. Furthermore, they found that the failure rates of flaky tests appear to differ

when executed in isolation. Of the 50 flaky tests that were reproduced in isolation, 19 had lower failure rates and 28 had higher ones. A Wilcoxon signed-rank test [97] demonstrated a statistically significant difference between the two samples, suggesting that the failure rates when executed within the test suite were significantly different from the failure rates when executed in isolation. This result shows that flaky tests appear to behave more consistently when executed in isolation and so this may be a good strategy if trying to reproduce the failure of a flaky test, but not its flakiness. Additionally, many tests previously called "non-order-dependent" actually do depend on the order and can fail with very different failure rates for different orders.

2.2 Cause and Categories of Flaky Test

Studies have examined and categorized the causes of flaky tests in general software projects, that is, not constrained to a particular platform or purpose. To that end, objects of study such as historical commit data [56], bug reports [93], developers' insights from flaky tests that they've previously repaired [21], pull requests [45] and execution traces [27] have been analyzed, resulting in a commonly used set of flakiness categories. Two of these studies consider subjects from the same source, namely, the Apache Software Foundation, yet interestingly present different findings [56, 93].

Intra-test family describes flakiness wholly internal to the test, in other words, stemming from issues isolated to the execution of the test code itself, or the direct code under test [68]. Concurrency is one of the common causes of flakiness. Test that invokes multiple threads interacting in an unsafe or unanticipated manner. Flakiness is caused by, for example, race conditions resulting from implicit assumptions about the ordering of execution, leading to deadlocks in certain test runs [56, 93]. Some tests use the result of a random data generator. If the test does not account for all possible cases, then the test may fail intermittently, e.g., only when the result of a random number generator is zero [21, 45]. Floating point operations can suffer from a variety of discrepancies and inaccuracies such as precision over and under flows, non-associative addition, etc., which if not properly accounted for can result in inconsistent test outcomes. One example is the comparison between the result of a floating point operation to an exact real value in an assertion [45, 56]. Certain flaky test assumes a particular iteration order for an unordered collection type object. Since no particular order is specified for such objects, tests that assume they will iterate in some fixed order will likely be flaky due to a variety of reasons such as the implementation of the collection class [27, 45, 56]. For scenarios where some of the valid output range falls outside of what is accepted in its assertions, the test is flaky because it does not account

for corner cases and thus it may intermittently fail when they arise. Tests specified with an upper limit on their execution time could be flaky tests. Since it's usually not possible to precisely estimate how long a test will take to run, by specifying an upper time limit a developer may run the risk of creating a flaky test, since it could fail on certain executions that are slower than anticipated [21, 27].

The inter-test family contains tests that are flaky with respect to the execution of other tests, for example, those with test order dependencies. Order dependency tests are those that depend on some shared value or resource that is modified by another test that impacts its outcome. In the case where the test run order is changed, these flaky tests may produce inconsistent outcomes, since the dependencies upon tests previously executed beforehand are broken [56, 93]. Some tests are built with improper handling of some external resources such as failing to release allocated memory. Improperly handled resources may cause flakiness in subsequently executed test cases that attempt to reuse that resource [27, 56, 93]. There are tests that become flaky solely because they are part of a test suite with a limited execution time. Intermittently fails, because it happens to be running once the test suite hits an upper time limit [21].

The flakiness of tests in the external family stems from factors outside of the test's control, such as the time taken to receive a response from a web server. One of the dominant groups of flaky tests are the ones that make an asynchronous call and do not explicitly wait for it to finish before evaluating assertions, typically using a fixed-time delay instead. Results may be inconsistent in executions where the asynchronous call takes longer than the specified time to finish, leading to the flakiness [56]. Tests calling non-deterministic I/O operations can be flaky due to the handling of input and output operations. For example, a test that fails when a disk has no free space or becomes full during file writing. The availability of a network connection, e.g., by querying a web server, can also cause flakiness. In the case where the network is unavailable or the required resource is too busy, the test may become flaky. Some test relies on local system time and they may be flaky due to discrepancies in precision and timezone, e.g., failing when midnight changes in the UTC timezone [21, 27, 45, 56]. Flakiness could be introduced because a test depends on some particular functionality of a specific operating system, library version, or hardware vendor. While such tests may produce a consistent outcome on a given platform, they are still considered flaky, particularly with the rise of cloud-based continuous integration services, where different test runs may be executed on different physical machines in a manner that appears non-deterministic to the user [21, 27, 93].

2.3 Existing Tools for Flaky Test Detection

Detection Tool Name	Approach
Flakiness Debugger [105]	Find the divergence of execution trace
DeFlaker [8]	Find difference in coverage between consecutive versions
FLASH [20]	Rerun with different random number seed
iDFlakies [46]	Rerun with randomized test order
NonDex [46]	Rerun tests with non-deterministic specifications
Shaker [81]	Rerun with randomly introduced CPU and memory task
FLAST [96]	Machine learning and treat test cases as text
FlakeFlagger [4]	Machine learning and extract features from source code
Flakify [25]	Machine learning without knowledge of source code

Table 2.1: Existing tools for flaky test detection and their approach

How to efficiently detect flaky tests is well-studied and many tools have been developed for such tasks. Table 2.1 provides a list of flakiness detection tools, along with a short description of their approach. The most straightforward techniques for automatically detecting flaky tests are based on repeatedly re-running them [8, 48]. The cost of this approach is high both in terms of time and computational power, which has necessitated more efficient approaches. At Google, developers create Flakiness Debugger [105] which inspects the execution trace of tests in their service and locates flakiness by finding the divergence of test results. A similar solution, DeFlaker [8], makes use of the difference in coverage between consecutive versions of the same software with the unstable outcome. An evaluation of this approach identified 96% of the flaky tests discovered by the traditional test re-execution approach [8]. A tool called FLASH [20] is developed to detect flaky tests specifically in Machine Learning applications. Another study presented iDFlakies, which is a framework based on executing test suites in randomized orders to detect order-dependent flaky tests [46] that is well-known for its effectiveness. NonDex [29] is another approach that randomizes the implementations of various Java classes with non-deterministic specifications to identify implementation-dependent flaky tests [79] and is able to find 60 flaky tests across 21 open-source Java projects [29]. Another study presented Shaker, which targets flaky tests of the asynchronous wait and concurrency categories by introducing CPU and memory tasks thus intervening in the order of events during regression tests [81]. Techniques from the field of machine learning have also been applied to flaky test detection, such as FlakeFlagger [4] and FLAST [96], and they considered the presence of particular keywords in test case source code or other general test characteristics, such as

execution time and line of code, as potential predictors of flaky tests [30, 71]. A new language model-based predictor for flaky tests called Flakify is recently developed. Flakify is able to statically predict flakiness without accessing the production code in a black-box manner [25].

Ziftci et al. [105] present, Flakiness Debugger, a novel technique to automatically identify the locations of the root causes of flaky tests on the code level to help developers debug and fix them. They study the technique on flaky tests across 428 projects at Google. Based on their case studies, the technique helps identify the location of the root causes of flakiness with 82% accuracy. The algorithm compares the execution traces of each failing run to all passing runs to find the first point of divergence in the control flow of the failing run from any of the passing runs, i.e. the point where a failing run's control flow has never been observed in a passing run. For all tests, they calculate the flakiness score $f(t)$ for t by checking how many times it flaked recently during its executions. Then, FD executes the test t a total of 50 times and collects dynamic execution traces for both passing and failing test executions.

Bell et al. [8] present the first extensive evaluation of rerunning failing tests and propose a new technique, called DeFlaker, that detects if a test failure is due to a flaky test without rerunning and with very low runtime overhead. DeFlaker monitors the coverage of the latest code changes and marks as flaky any newly failing test that did not execute any of the changes. They deployed DeFlaker live, in the build process of 96 Java projects on TravisCI, and found 87 previously unknown flaky tests in 10 of these projects. They also ran experiments on project histories, where DeFlaker detected 1,874 flaky tests from 4, 846 failures, with a low false alarm rate (1.5%). DeFlaker had a higher recall (95.5% vs. 23%) of confirmed flaky tests than Maven's default flaky test detector. Flaky tests can disrupt developers' workflows because it is difficult to know immediately if a test failure is a true failure or a flake.

Another framework, iDFlakies [46], is developed to automate experimentation to detect and partially classify flaky tests using their tool for Maven-based Java projects with JUnit tests. They have applied the framework to 683 projects. They provide a dataset of 422 flaky tests that they then use for their study on flaky tests. From their dataset, 50.5% of flaky tests are Order-Dependent, while 49.5% are Non-Order-Dependent, based on the observed runs. They also find that running the random-class-method configuration can detect the most flaky tests overall.

Gyori et al. [29] present NonDex, a tool for detecting and debugging wrong assumptions on Java APIs. Some APIs have underdetermined specifications to allow implementations to achieve different goals, e.g., to optimize performance. When clients of such APIs as-

sume stronger-than-specified guarantees, the resulting client code can fail. For example, HashSet's iteration order is underdetermined, and code assuming some implementation-specific iteration order can fail. NonDex helps to proactively detect and debug such wrong assumptions by performing detection using random explorations on different behaviours of underdetermined APIs during test execution. When a test fails during exploration, NonDex searches for the invocation instance of the API that caused the failure. During their experiments with the NonDex Maven plugin, they detected 21 new bugs in eight Java projects from GitHub, and, using the debugging feature of NonDex, they identified the underlying wrong assumptions for these 21 new bugs and 54 previously detected bugs. They opened 13 pull requests; developers already accepted 12, and one project changed the continuous-integration configuration to run NonDex on every push. In general, NonDex tool is developed to help in detecting and debugging wrong assumptions on underdetermined APIs in Java.

SHAKER [81] is a lightweight technique to improve the ability of ReRun to detect flaky tests. SHAKER adds noise in the execution environment (e.g., it adds stressor tasks to compete for the CPU or memory). It builds on the observations that concurrency is an important source of flakiness and that adding noise in the environment can interfere with the ordering of events and, consequently, influence the test outputs. They conducted experiments on a data set with 11 Android apps. The results are very encouraging. SHAKER discovered many more flaky tests than ReRun (95% and 37.5% of the total, respectively) and discovered these flaky tests much faster. In addition, SHAKER was able to reveal 61 new flaky tests that went undetected in 50 re-executions with ReRun.

Verdecchia et al. [96] present and evaluate FLAST, an approach designed to statically predict test flakiness. FLAST leverages vector-space modelling, similarity search, dimensionality reduction, and k-Nearest Neighbor classification to timely and efficiently detect test flakiness. In order to gain insights into the efficiency and effectiveness of FLAST, they conduct an empirical evaluation of the approach by considering 13 real-world projects, for a total of 1,383 flaky and 26,702 non-flaky tests. They carry out a quantitative comparison of FLAST with the state-of-the-art methods to detect test flakiness, by considering a balanced dataset comprising 1,402 real-world flaky and as many non-flaky tests. From the results, they observe that the effectiveness of FLAST is comparable with the state-of-the-art while providing considerable gains in terms of efficiency. In addition, the results demonstrate how by tuning the threshold of the approach FLAST can be made more conservative, so to reduce false positives, at the cost of missing more potentially flaky tests. The collected results demonstrate that FLAST provides a fast, low-cost and reliable approach that can be used to guide test rerunning, or to gate the inclusion of new potentially flaky tests. Flaky tests that pass FLAST's filtering can still be detected by dynamic approaches like

DeFlaker or even Rerun, but with much fewer resources. Also, this combination of FLAST with dynamic approaches is an important objective for future work.

Pinto et al. [71] conducted a similar study and evaluated the performance of various machine learning algorithms to solve flaky test detection. They constructed a data set of flaky and non-flaky tests by running every test case, in a set of 64k tests, 100 times (6.4 million test executions). They then used machine learning techniques on the resulting data set to predict which tests are flaky from their source code. There are three interesting findings. First, they were able to find six projects with flaky tests, and a total of 86 flaky tests. More interestingly, however, is the fact that 55% of these flaky tests failed just once, meaning that the 100 threshold might have limited the observation of flaky tests. Second, they observed that the five machine learning algorithms used had good performance in distinguishing flaky from non-flaky tests. In particular, Random Forest had the best precision (0.99), while the Support Vector Machine slightly outperformed Random Forest in terms of recall (0.92 vs 0.91). Third, in terms of the features used in the classifiers for improving performance, they noticed that, for both Random Forest and Support Vector Machine, perhaps surprisingly, most of the features in the classifier did not have a visible impact on the results. Finally, regarding the vocabulary of flaky tests, they noticed that words such as job, table, or action (which are often associated with remote work) are among the features with the highest information again. Overall, their results provide initial yet strong evidence that static analysis of flaky tests is effective.

Alshammari et al. [4] conducted a very large empirical study looking for flaky tests by rerunning the test suites of 24 projects 10,000 times each, and found that even with this many reruns, some previously identified flaky tests were still not detected. They then proposed FlakeFlagger, a novel approach that collects a set of features describing the behaviour of each test and then predicts tests that are likely to be flaky based on similar behavioural features. They found that FlakeFlagger correctly labelled as flaky at least as many tests as a state-of-the-art flaky test classifier, but that FlakeFlagger reported far fewer false positives. This lower false positive rate translates directly to saved time for researchers and developers who use the classification result to guide more expensive flaky test detection processes. Evaluated their dataset of 23 projects with flaky tests, FlakeFlagger outperformed the prior approach (by F1 score) on 16 projects and tied on 4 projects. Compared to a state-of-the-art flaky test classifier, FlakeFlagger had a similar recall (74% vs 72%), but significantly better precision (60% vs 11%). This improvement translates to a significant reduction in the number of misdiagnosed flaky tests, dramatically reducing the number of (possibly) flaky tests that need to be further investigated. Our results indicate that this approach can be effective for identifying likely flaky tests prior to running time-consuming flaky test detectors.

Fatima et al. [25] pointed out that the state-of-the-art ML-based flaky test case predictors rely on pre-defined sets of features that are either project-specific, i.e., inapplicable to other projects, or require access to production code, which is not always available to software test engineers. Moreover, given the non-deterministic behaviour of flaky test cases, it can be challenging to determine a complete set of features that could potentially be associated with test flakiness. Therefore, they propose Flakify, a black-box, language model-based predictor for flaky test cases. Flakify relies exclusively on the source code of test cases, thus not requiring (a) access to production code (black-box), (b) rerun test cases, (c) pre-define features. To this end, they employed CodeBERT, a pre-trained language model, and fine-tuned it to predict flaky test cases using the source code of test cases. They evaluated Flakify on two publicly available datasets for flaky test cases and compared their technique with the FlakeFlagger [4] approach, the best state-of-the-art ML-based, white-box predictor for flaky test cases, using two different evaluation procedures: (1) cross-validation and (2) per-project validation, i.e., prediction on new projects. Flakify achieved F1-scores of 79% and 73% on the FlakeFlagger dataset using cross-validation and per-project validation, respectively. Similarly, Flakify achieved F1-scores of 98% and 89% on the IDoFT dataset using the two validation procedures, respectively. Further, Flakify surpassed FlakeFlagger by 10 and 18 percentage points (pp) in terms of precision and recall, respectively, when evaluated on the FlakeFlagger dataset, thus reducing the cost bound to be wasted on unnecessarily debugging test cases and production code by the same percentages (corresponding to reduced rates of 25% and 64%). Flakify also achieved significantly higher prediction results when used to predict test cases on new projects, suggesting better generalizability over FlakeFlagger. Their results further show that a black-box version of FlakeFlagger is not a viable option for predicting flaky test cases.

2.4 Repair and Mitigate Flaky Test

An early study proposed and evaluated a lightweight alternative to mitigating test order dependencies via process isolation, based on dynamically re-initializing modified values following test runs, finding it to introduce an average time overhead of only 34% compared to 618% [6]. In the context of test suite parallelization, one study proposed and evaluated schedulers for making the most of available CPUs when executing test suites with known and specified order-dependent tests, reporting an average speedup of up to 7 times compared to a regular test suite run without breaking any test order dependencies [7]. Another source examined how best to configure concurrent execution when faced with order-dependent tests, with the main finding being that ensuring consistent

test outcomes appears to be a trade-off with achieving the fastest test runs [10]. One experiment demonstrated the applicability of an algorithm for satisfying broken dependencies following rescheduling by prioritization, selection or parallelization, reducing the number of failed order-dependent tests in developer-written test suites by between 79% and 100% [47]. A study into mitigating flaky test coverage in mutation testing found that repeating and isolating test runs during the mutant killing phase reduced the number of non-deterministically killed mutants by 79% [78].

Depending on the cause of flakiness, the options for mitigating and repairing flaky tests may vary. Based on the findings from historical commits in Apache Software Foundation projects and surveying Mozilla developers, two studies [21, 55] concluded that the most common type of fix for flaky tests of the asynchronous wait category involved a `waitFor` method or its equivalent. For order-dependent tests, the most common fixes are to eliminate the dependency between its victim and brittle in a test's allocation and de-allocation methods by techniques such as creating a duplicate instance of some shared resource [21, 55]. The iFixFlakies framework [80] was presented as the automatic repair of order-dependent flaky tests, which uses program statements from elsewhere in the test suite, and has generated fixes for 110 order-dependent tests, with 21 accepted by developers through pull requests. Another study presented a template-based repair technique called DexFix [103] for automatically repairing implementation-dependent flaky tests, such as those detected by the previously mentioned detection tool NonDex [29]. By applying this technique, 119 of 275 flaky tests have been repaired across 37 open-source Java projects, and the generated patches of 74 have been submitted as pull requests to their respective repositories and merged by their developers [79, 103].

Studies have provided insights into repairing various kinds of flaky tests by considering previously applied fixes in general software projects. Luo et al. [55] examined commits that repaired flaky tests within projects from the Apache Software Foundation. Eck et al. [21] surveyed Mozilla developers about flaky tests they'd previously fixed. In terms of where these were applied, figures range from between 71% and 88% exclusively to test code [21, 45, 55]. The remainder was applied to either the code under test, both the test code and the code under test or elsewhere, such as within configuration files. This finding indicates that the origin of test flakiness may not exclusively be test code, and in the cases where fixes were applied to non-test code, suggests that a flaky test has, possibly indirectly, led a developer to discover and repair a bug in the code under test. This provides an additional argument against ignoring flaky tests. With regards to repairing flaky tests of the asynchronous wait category, the most common fix involved introducing a call to Java's waitFor, or Python's await, or modifying an existing such call, accounting for between 57% and 86% of previous cases. The waitFor method blocks the calling thread

until a specific condition is satisfied, or until a time limit is reached. This would be used to explicitly wait until an asynchronous call has fully completed before evaluating assertions, thus eliminating any timing-based flakiness. Similarly, Luo et al. [55] identified fixes regarding the addition or modification of a fixed time delay, such as via a call to Java's sleep, to make up 27% of historical repairs. Compared to waitFor, sleep is a less reliable solution, since the specified time delay can only ever be an estimate of the upper limit of the time taken for the asynchronous call to complete in any given test run. To that end, they identified that 60% of such repairs increased the time delay, suggesting that the developers believed their estimate to be too low or perhaps too unreliable across different machines. A study by Malm et al. [58] found sleep-type delays to generally be more common than waitFor, via automatic analysis of seven projects written in the C language. Another solution was to simply reorder the sequence of events such that some useful computation is performed after making an asynchronous call instead of quiescently waiting for it to complete, accounting for between 3% and 13% of cases. Such a change does not actually address the root problem, but may be preferable to simply blocking the calling thread.

Lu et al. [54] studied real-world concurrency bugs, and found that most of the concurrency bugs belong to order or atomicity violations. For fixing flaky tests of the concurrency category, the most common repair, according to Eck et al., again pertained to the introduction or modification of a call to waitFor or await, with a prevalence of 46%. Adding locks to ensure mutual exclusion between threads accounted for between 21% and 31% of fixes for flaky tests in this category. Between 9% and 26% of historical repairs involved modifying concurrency guard conditions, such as the conditions for controlling which threads may enter which regions of code at any one time. Making code more deterministic by eliminating concurrency and enforcing sequential execution made up between 5% and 25% of fixes. An additional type of repair as identified by Luo et al. [55] consisted of modifying test assertions, and nothing else, to account for all possible valid behaviours in the face of the non-determinism permitted by the concurrent program, describing 9% of previous repairs. Ensuring proper setup and teardown procedures between test runs was the most common fix for order-dependent tests according to Luo et al. [55], accounting for 74% of cases. Another fix was to explicitly remove the test order dependency by, for instance, making a copy of the associated shared resource (e.g., an object or directory), making up between 16% and 100% of repairs. Another repair was to merge tests into a single, independent test by, for example, simply copying the code of one test into another, which described 10% of fixes for order-dependent tests according to Luo et al. [55].

As part of an empirical evaluation into flakiness specific to user interface tests, Romano et al. [73] examined 235 developer-repaired flaky user interface tests across a sample of web

and Android applications. They categorized the type of repair applied into five categories. The first category, delay, accounted for 27% of cases and was subcategorized into repairs involving the addition or increase of a fixed time delay or those relating to an explicit waiting call, as described previously. The second category, dependency, is related to repairs regarding an external dependency, such as fixing an incorrectly called API function or changing the version of a library. They placed 8% of flaky tests into this category. The final three categories were refactored test, disabled features (specifically disabling animations in the user interface, which was found to be a significant cause of flakiness), and remove test. These represented 32%, 2% and 31% of repairs, respectively, the latter category not being a true repair, but simply eliminating the flaky test from the test suite.

Besides manual repairs, Shi et al. [80] proposed iFixFlakies, a framework for automatically fixing order-dependent tests. The key insight in iFixFlakies is that test suites often already have tests, which they call helpers, whose logic resets or sets the states for order-dependent tests to pass. iFixFlakies searches a test suite for helpers that make the order-dependent tests pass and then recommends patches for the order-dependent tests using code from these helpers. their evaluation of 110 truly order-dependent tests from a public dataset shows that 58 of them have helpers, and iFixFlakies can fix all 58. They opened pull requests for 56 order-dependent tests (2 of 58 were already fixed), and developers have already accepted pull requests for 21 of them, with all the remaining ones still pending. The recommended patches are effective, with 69.5% of them having just one statement. Also, iFixFlakies is efficient, requiring only 238 seconds on average to produce the first patch for an order-dependent test with a helper.

Library developers can provide classes and methods with approximated specs that are underdetermined allowing flexibility in future implementations which could potentially lead to flakiness. Library users may write code that relies on a specific implementation rather than on the specification, e.g., assuming mistakenly that the order of elements cannot change in the future. Prior work proposed the NonDex [29] approach that detects such wrong assumptions. Zhang et al. [103] present a novel approach, called DexFix, to repair wrong assumptions on underdetermined specifications in an automated way. They run the NonDex tool on 200 open-source Java projects and detect 275 tests that fail due to wrong assumptions. The majority of failures are from iterating over HashMap/HashSet collections and the get DeclaredFields method. They provide several new repair strategies that can fix these violations in both the test code and the main code. DexFix proposes fixes for 119 tests from the detected 275 tests. They have already reported fixes for 102 tests as GitHub pull requests: 74 have been merged, with only 5 rejected, and the remaining pending.

2.5 Summary

Flaky tests are a common phenomenon in industry-scale software systems and affect development unfavourably. They cause unnecessary debugging time for the practitioners and violates the underlying deterministic assumption of many testing techniques. There are many reasons behind flaky tests such as asynchronous wait, concurrency issues and resource allocation conflicts which lead to the definition of different categories of flaky tests. Many researchers also proposed ways of repairing and mitigating flakiness in the test suites and some automated frameworks are built and evaluated to address specific types of flaky tests. This thesis contributes to fill the research gap between the flaky test detection and flaky test categorization. It proposes a novel machine learning-based framework for fast and accurate prediction on given flaky tests. The existing category-specific fixes for flaky tests can then be easily applied.

Chapter 3

Framework Architecture

In this chapter, the target of the proposed framework, FlaKat, is presented along with its architecture. Section 3.1 lists the workflow of FlaKat and the duties of its four phases. Section 3.2 lists the goals of the framework that guide the creation of its architecture and decisions on its realization.

3.1 FlaKat: Workflow

The workflow of FlaKat follows the conventional design of frameworks incorporating machine learning algorithms. The input of the framework is a list of known flaky tests which consist of of flaky test names, URLs to their online repositories and categorical labels. Phase I of the FlaKat parses the raw Java test codes from the target URL and handles the preprocessing. In Phase II various source code representation techniques are applied to convert the raw code into vector embeddings and their dimensionality is reduced in Phase III. Lastly, Phase IV makes the final prediction with sampling techniques applied to address the imbalance between the categories in the dataset. Fig. 3.1 illustrates the workflow of the entire FlaKat and details of each phase are provided below.

• **Phase I: Parsing and Preprocessing**

In Phase I, FlaKat receives the input data which consists of the full name of the flaky test, the URL of the project where the flaky test belongs, the hash of the commit when flakiness is detected and the category of flakiness. Manually downloading the source code and locating the flaky test would be extremely time-consuming and impractical. Thus, an automated parser is necessary. The parser first clone the target open-source project

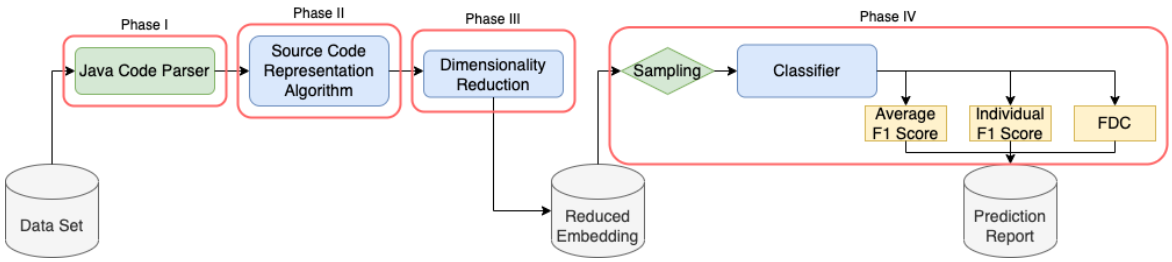


Figure 3.1: Workflow of FlaKat and its four phases

on GitHub from its online repository to a local directory. Then the correct version for reported flakiness is checkout. Once the correct source code is obtained, the parser then collected all flaky tests in the projects in the test file. For the convenience of vectorization in the next phase, the raw test codes are flattened into single lines of strings. After Phase I finishes, we obtain a list of flaky tests including the raw test code, the name of the test function and the category of the test. Phase I must be executed at least once so that FlaKat can acquire valid input with the right format for the following phases.

- **Phase II: Source Code Vectorization**

Raw test codes themselves cannot be directly processed as the input of machine learning classifiers and Phase II of FlaKat handles the conversion from flaky test codes to their vector representations. Depending on the type of vectorization technique, the raw test case string is converted into the same number of vectors of varying floating points. The range of the values within the vector depends on the specific implementation of the vectorization algorithm. In this research, the expense of time for the conversion is not part of the requirement and thus is not recorded for future evaluation. There are pre-trained models available for certain vectorization algorithms and they are used. If not, the models are trained using the raw test cases parsed in Phase I. After Phase II successfully finishes, we obtain a list of flaky tests including vectors of floating points representing the original raw code of the test case, the name of the test case and the category of the test.

- **Phase III: Dimensionality Reduction**

The vector representation of flaky tests has a high number of features and using them to train and test machine learning models are impractical in terms of time. In Phase III, the vector embeddings of raw Java test cases are reduced to low dimensions. Different techniques are explored and tuning is applied to those with tunable hyperparameters. Besides keeping the training time of classifiers to a realistic scale, the reduction to 2D can also help visualize the flaky test data points. Observation based on these 2D projections can show whether distinct clusters of flaky tests from different categories exist or not. The

analysis is performed at end of this phase both qualitatively and quantitatively to determine which dimensionality reduction technique best preserve the local and global structure of vector embeddings of flaky tests. At the end of Phase III, the reduced embedding of flaky tests is obtained along with their names and categories.

• **Phase IV: Sampling and Prediction**

In Phase IV, the framework makes the prediction on the category of given flaky tests which is a critical phase for achieving optimal prediction accuracy. Based on the existing knowledge [68], the ratio between different categories of flaky tests could be drastically different. Thus sampling techniques are required to address the such imbalance. Combined with cross-validation, the sampling is applied to the training dataset after the split of the dataset of reduced embedding of flaky tests. Over-sampling is be applied first followed by the under-sampling technique.

The balanced dataset is then be used to train the machine-learning models. Their accuracy is be recorded with the corresponding configuration for tuning and analysis. Another round of evaluation is conducted in the end to determine which machine learning classifier yields the optimal results.

3.2 Flakat: Research Goals

FlaKat is proposed to address the gap between the existing flaky test detection solutions and category-specific flakiness repairs. To achieve this goal, several qualities are required and listed below:

- Efficient — The framework should be able to predict the category of flaky tests without re-running the test suites.
- Accurate — The framework should be able to achieve decent accuracy in the prediction that can later help repairing flakiness.
- Extensible — The framework should be able to divide the key components within the framework into modules allowing exploration of different implementations and future upgrades.

Incorporating machine learning algorithms into the identification of flaky tests has proved to be effective in recent studies [4, 25, 71, 96]. The vocabulary-based approaches

firstly vectorize test case source code similar to natural language. Then, use machine learning classifiers, such as Random Forest, to predict whether a given test case is flaky or not [71, 96]. Other feature-based approaches consider more factors related to the behaviour of test cases and extract additional source code features, such as the number of lines and test execution time [4]. Both approaches can perform flakiness predictions with good performance with low overhead. Vocabulary-based is chosen for FlaKat. The first reason is that feature-based approaches require more knowledge of the test cases that can only be acquired by repeated executions. Secondly, there is plenty of research available in source code representation and exploring their efficiency in capturing flakiness characteristics seems promising.

A generic repair strategy for all types of flaky tests is not available yet. However, the current literature already has plenty of flaky test repair techniques for addressing specific types of flaky tests. The concurrency-related flakiness can be mitigated by adding `waitFor` methods for the asynchronous wait blocks [21, 55]. Automatic tools have also been developed for order-dependent and implementation-dependent flaky tests [80, 103] and achieved successful fixes in real-life software projects to improve the reliability of their test suites. The accuracy of prediction on the Implementation-Dependent and Order-Dependent flaky test has higher priority compared to other categories.

Thus, the proposed framework, FlaKat, adopts the vocabulary-based approach and uses machine learning algorithms for fast and accurate predictions for categorical classifications and group test cases into categories such as order-dependent and implementation-dependent. Compared to the existing tools [4, 96], developers can gain more insight into the nature of the flaky tests and can easily find the corresponding repair techniques and boost their productivity while also avoiding the cost of long runtime from repeated test execution.

3.3 Summary

The FlaKat framework adopts a machine learning approach and aims to achieve efficient and accurate prediction in the category of flaky tests. The input of the framework is a list of known flaky tests that contains their version and location on Github and their categories and the raw Java test cases is parsed in Phase I. These test cases are converted into vector embedding in Phase II and then reduce to a lower dimension in Phase III. The last Phase IV handles sampling on the training dataset split from the reduced embedding and then trains the machine learning model to make the prediction on the category of flaky tests.

Chapter 4

Framework Realization

The realization of FlaKat aims to achieve the goals listed in Sec. 3.2 to find the possible implementation for components mentioned in Chapter 3. Parsing and pruning in Phase I are implemented using the Git python package. In Phase II of the FlaKat, three source code vectorization techniques, doc2vec [50], code2vec [3], and tf-idf [40], are used to vectorize the Java flaky tests after parsing and preprocessing. Dimension reduction techniques, PCA [38], LDA [89], Isomap [87], t-SNE [94] and UMAP [59], are applied in Phase III to the vector embeddings. The reduced embeddings are analyzed both qualitatively and quantitatively to select the one that best preserves the local and global structure of flaky test vector embeddings. Evaluation of the machine learning classifier used in Phase IV (KNN [18], SVM [17] and Random Forest [33]) is based on two metrics: F_1 score and FDC. In the end, a final conclusion is made regarding which combination of code vectorization method, dimensionality reduction technique, and machine learning classifier yields the best performance in flaky test categorization.

4.1 Java Test Parser

Phase I of FlaKat handles the **parsing and preprocessing** of the input flaky tests. The test cases that FlaKat used for categorization are from open-source Java projects accessible on Github. Manually downloading their source code to the local machine and locating the labelled flaky tests is unrealistic in terms of effort and time required. Thus FlaKat requires an automated parser to handle the task of cloning the source at the exact commit of reporting flakiness. Additionally, the parser should also be able to locate the flaky test cases given test names and remove the unnecessary codes.

The flaky tests dataset FlaKat uses is the International Dataset of Flaky Test (IDoFT) [29, 80] which includes the full name of the flaky test, the URL of the project where the flaky test belongs, the hash of the commit when flakiness is detected and the category of flakiness. To parse the flaky Java test cases, FlaKat uses the external GitPython library [26]. It is a python library used to interact with git repositories. It provides abstractions of git objects for easy access to repository data and additionally allows FlaKat to access the git repository more directly using a pure python implementation.

For FlaKat, the input already contains all required information to locate the flaky tests in their git repositories. The first step to parse the input flaky tests is to download them to the local machine and it is done using the *clone_from* function supported by *git.Repo* with the URL of the project. Then the specific commit provided in the input will be checkout to ensure the test codes are in the exact commit when flakiness is reported. This step is necessary because the developers from certain projects are already aware of the flakiness and repaired the flaky tests already. Then the raw test cases will be extracted based on their file path derived from their fully-qualified test name. All the extracted raw test cases will be flattened into one string and stored in csv file with its test name and category label.

4.2 Source Code Representation Algorithms

The parsed raw Java test cases are then converted into vector embeddings by *source code vectorization* as Phase II of FlaKat. Three well-known vectorization methods: doc2vec, code2vec and tf-idf, are selected for FlaKat to explore how well their embeddings reflect the characteristic of flakiness. The realization details of each source code representation algorithm are as follow.

4.2.1 Doc2vec

The first vectorization method selected for FlaKat is doc2vec [50]. Java source can be seen as texts with a special format and when it comes to texts and one of the most common fixed-length features is bag-of-words which is supported in word2vec [61]. Despite their popularity, bag-of-words features have a weakness in that they lose the ordering of the words and they also ignore the semantics of the words. In the case of flaky tests, the order of keywords could reflect the nature of flakiness [55]. Doc2vec algorithm can overcome these problems. It is an unsupervised algorithm that learns fixed-length feature representations from variable-length pieces of text, such as sentences, paragraphs, and

documents. The algorithm represents each document by a dense vector which is trained to predict words in the document. Empirical results [50] show that Paragraph Vectors outperform bag-of-words models and achieve new state-of-the-art results on several text classification and sentiment analysis tasks. Similar to word2vec, doc2vec also supports two models, Distributed Memory Models of Paragraph Vector (PV-DM) and Distributed Bag of Words of Paragraph Vector (PV-DBOW) and PV-DM alone usually works well for most tasks with state-of-art performances.

The open-source gensim library [19] for documents representation has the implementation of doc2vec as a Python method and FlaKat will incorporate that. The PV-DM algorithm will be used and the model will be trained using the downloaded raw Java test cases from Phase I.

4.2.2 Code2vec

Code2vec [3] is selected as the second source code vectorization technique. Compare to doc2vec which is designed mostly for natural language processing, code2vec targets programming language. It is a neural model representing snippets of code as continuously distributed vectors - code embedding which enables modelling of correspondence between code snippets as well as natural and effective labelling. Predicting the category of flaky tests is an ideal task that fits its purpose.

The neural network architecture of code2vec uses a representation of code snippets that leverages the structured nature of source code and learns to aggregate multiple syntactic paths into a single vector. This ability is fundamental for the application of deep learning in programming languages. By analogy, word embeddings in natural language processing (NLP) started a revolution in the application of deep learning for NLP tasks. The input to the model is a code snippet and a corresponding tag, label, caption, or name. This label expresses the semantic property that the network wishes to model, for example, a tag that should be assigned to the snippet, or the name of the method, class, or project that the snippet was taken from. Let C be the code snippet and L be the corresponding label or tag. The underlying hypothesis is that the distribution of labels can be inferred from syntactic paths in C. The model, therefore, attempts to learn the label distribution, conditioned on the code: $P(L|C)$. During the process of learning code vectors, a parallel vocabulary of vectors of the labels is learned. When using the model to predict method names, the method-name vectors provide surprising semantic similarities and analogies.

The implementation of code2vec used in FlaKat is pulled from their Github page [15] on Jan 2022. Their pre-trained model is also available on the same repository and will

be used saving us time and effort in finding a large set of test functions. The default configuration will be used for converting flaky tests into vector embedding.

4.2.3 Term Frequency — Inverse Document Frequency

Term Frequency — Inverse Document Frequency (tf-idf) [40] is a technique to quantify words in a set of documents. Generally computing a score for each word can signify its importance in the document and corpus. This method is a widely used technique in Information Retrieval and Text Mining. Additionally, its effectiveness in reflecting flakiness characteristics has been proved in [96].

Term Frequency measures the frequency of a word in a document. This highly depends on the length of the document and the generality of the word, for example, a very common word such as "was" can appear multiple times in a document. But if two documents have 100 words and 10,000 words respectively, there is a high probability that the common word "was" is present more in the 10,000 worded document and longer documents are not necessarily more important than shorter ones. Thus, normalization on the frequency value is performed, the frequency is divided by the total number of words in the document. However, words which are most common such as 'is', and 'are' will have very high values, giving those words very high importance. But using these words to compute the relevance produces bad results. These kinds of common words are called stop-words. Although the stop words would be removed later in the preprocessing step, finding the presence of the word across the documents and somehow reducing their weightage is ideal.

Document Frequency measures the importance of documents in a whole set of the corpus. This is very similar to TF but the only difference is that TF is the frequency counter for a term t in document d, whereas DF is the count of occurrences of term t in the document set N. In other words, DF is the number of documents in which the word is present. DF counts one occurrence if the term is present in the document at least once, the number of times the term is presented is not required. IDF is the inverse of the document frequency which measures the informativeness of term t. When IDF is calculated, it will be very low for the most occurring words such as stop words because they are present in almost all of the documents, and N/df will give a very low value to that word. This finally gives a relative weightage.

There is an existing mature implementation of tf-idf in Scikit-learn [88] and that will be incorporated into FlaKat. No training or learning is required for tf-idf algorithm and all configurable parameters remain default values except for the number of features that determines the dimensionality of the generated vector embedding.

4.3 Dimensionality Reduction Techniques

In Phase III of FlaKat, several *dimensionality reduction* techniques will be applied to the three vector embeddings generated by the previously mentioned models. As stated in the workflow, the target of Phase III is 1) to ensure the training of the machine learning classifier in Phase IV can finish in several minutes and 2) to visualize the data points of the flaky test so that we can gain more insight on the clustering and distribution of flaky tests in vector space. For consistency, all reduced embedding will be in two dimensions.

The raw Java test cases are then converted into vector embeddings via doc2vec, code2vec or tf-idf with different dimension reduction techniques applied to them. The techniques considered in Phase III of FlaKat are PCA [38], LDA [89], Isomap [87], t-SNE [94], and UMAP [59]. Unlike PCA and LDA, the Isomap, t-SNE, and UMAP reductions all have tunable hyperparameters that affect the result of the dimension reduction. The results from different numbers of neighbours to consider for Isomap, various perplexity for t-SNE, and numerous combinations of the number of neighbours and minimum distance between valid neighbours for UMAP, are collected as scatter plots. The results with the cleanest clustering are selected for comparison with other reduction techniques both qualitatively and quantitatively.

4.3.1 Principal Component Analysis

Principal Component Analysis (PCA) [38] is a technique for reducing the dimensionality of such datasets, increasing interpretability but at the same time minimizing information loss. It does so by creating new uncorrelated variables that successively maximize variance. Finding such new variables, the principal components, reduces to solving an eigenvalue/eigenvector problem, and the new variables are defined by the dataset at hand, not a priori, hence making PCA an adaptive data analysis technique. It is adaptive in another sense too, since variants of the technique have been developed that are tailored to various different data types and structures.

For FlaKat, the PCA decomposition supported by Scikit-learn is sufficient [70]. There aren't any tunable parameters for PCA that significantly affects its reduction and the default setting will be applied in FlaKat.

4.3.2 Linear Discriminant Analysis

The general Linear Discriminant Analysis (LDA) [89] approach is very similar to a PCA, but in addition to finding the component axes that maximize the variance of the data (PCA), it is additionally interested in the axes that maximize the separation between multiple classes (LDA). Another major difference between PCA and LDA is that PCA can be described as an "unsupervised" algorithm since it "ignores" class labels and its goal is to find the principal components that maximize the variance in a dataset. LDA, however, is "supervised" and computes the linear discriminants that will represent the axes that maximize the separation between multiple classes.

For FlaKat, the LDA decomposition supported by Scikit-learn is also sufficient [49] and the default setting will be used.

4.3.3 Isometric Mapping

Isometric Mapping (Isomap) [87] is a technique that combines several different algorithms, enabling it to use a non-linear way to reduce dimensions while preserving local structures. It first uses a KNN approach to find the k nearest neighbours of every data point. Once the neighbours are found, the neighbouring graph is constructed where points are connected to each other if they are each other's neighbours. Then it computes the shortest path between each pair of data points and uses multidimensional scaling to compute lower-dimensional embedding while balancing the relationship between local and global structures.

Similar to PCA and LDA, Scikit-learn library supports Isomap and can be applied in FlaKat directly [35]. The number of neighbours to consider for each point in finding the neighbour will be explored and the best reduction by observation for each vector embedding will be used for later analysis.

4.3.4 T-Distributed Stochastic Neighbor Embedding

The t-Distributed Stochastic Neighbor Embedding (t-SNE) [94] is an unsupervised, non-linear technique that works well with dimensionality reduction and data visualization. It calculates a similarity measure between pairs of instances in the high dimensional space and in the low dimensional space. It then tries to optimize these two similarity measures using a cost function.

The existing implementation from Scikit-learn will be used in FlaKat [85]. T-SNE allows users to adjust its perplexity which is related to the number of nearest neighbours

that is used in other manifold learning algorithms such as Isomap[87]. Different perplexities will be used and the optimal result will be selected for analysis later.

4.3.5 Uniform Manifold Approximation and Projection

Uniform Manifold Approximation and Projection (UMAP) [59] is a dimension reduction technique that can be used for visualization similarly to t-SNE, but also for general non-linear dimension reduction which is largely based on manifold theory and topological data analysis. It uses local manifold approximations and patches together with their local fuzzy simplicial set representations to construct a topological representation of the high dimensional data. Given some low-dimensional representation of the data, a similar process can be used to construct an equivalent topological representation. UMAP then optimizes the layout of the data representation in the low dimensional space, to minimize the cross-entropy between the two topological representations.

The UMAP implementation is available online [92] and can be imported into FlaKat. However, its superior performance compared to t-SNE is achieved under the assumption that the data is uniformly distributed on the Riemannian manifold. Whether such an assumption holds true for flaky test data points in vector space is currently unknown. Compare to Isomap and t-SNE, the reduction from UMAP is affected by both numbers of neighbours to consider and the minimum distance to consider. Various combinations of the two parameters will be used and only the one that yields the best result will be selected.

4.4 Oversampling and Undersampling Techniques

Several studies on the occurrence of the flaky test have shown that different categories of flaky tests occur at different rates. Thus imbalance between different categories of flaky tests is inevitable and widely exists in all the current flaky test dataset [68, 104, 79]. A severe imbalance of the classes can be challenging to model and may require the use of specialized techniques. The minority class is harder to predict because there are few examples of this class, by definition. This means it is more challenging for a model to learn the characteristics of examples from this class and to differentiate examples from this class from the majority class (or classes). The abundance of examples from the majority class (or classes) can swamp the minority class. Most machine learning algorithms for classification predictive models are designed and demonstrated on problems that assume an equal distribution of classes. This means that a naive application of a model may focus

on learning the characteristics of the abundant observations only, neglecting the examples from the minority class that is, in fact, of more interest and whose predictions are more valuable. Thus, both *oversampling and undersampling* techniques will be applied to the training set in Phase IV.

4.4.1 Synthetic Minority Oversampling Technique

Synthetic Minority Oversampling Technique (SMOTE) [11] will be used to oversample the minority categories of flaky tests. SMOTE works by selecting examples that are close to the feature space, drawing a line between the examples in the feature space and drawing a new sample at a point along that line. Specifically, a random example from the minority class is first chosen. Then k of the nearest neighbours for that example is found (typically k=5). A randomly selected neighbour is chosen and a synthetic example is created at a randomly selected point between the two examples in the feature space. A general downside of the approach is that synthetic examples are created without considering the majority class, possibly resulting in ambiguous examples if there is a strong overlap for the classes.

The implementation supported by Imbalanced-learn Python library will be imported and used in FlaKat [82].

4.4.2 Tomek Link

Tomek Links [90] is an undersampling technique designed to reduce the majority class size. The method will scan through the provided dataset and removes data points when it belongs to the majority class and also its nearest neighbour belongs to a minority class. After the iteration finishes, the remaining dataset can obtain better clustering with a clean separation between the majority class and minorities.

Similar to SMOTE, Imbalanced-learn also provides an implementation of Tomek Link and can be used in FlaKat immediately [90].

4.5 Machine Learning Classifiers

At the second part of Phase IV, the training dataset is balanced and ready to be processed by *machine learning classifiers*. There are plenty of machine learning models available in Python that can process the vector embedding of flaky tests and make predictions. Several

popular classifiers from the Scikit-learn library were selected. First, K-Nearest Neighbour (KNN) is considered here because it was the classifier adopted in FLAST [96] and has been shown to have high performance. Another algorithm with outstanding performance is Random Forest, which has been shown to surpass KNN and other models in flaky/non-flaky prediction [71]. Lastly, Support Vector Machine (SVM) is also considered since it is well-known for the non-linear classification of natural data point clusters.

4.5.1 K-Nearest Neighbours

The K-Nearest Neighbours algorithm (KNN) [18] is a non-parametric, supervised learning classifier, which uses proximity to make classifications or predictions about the grouping of an individual data point. While it can be used for either regression or classification problems, it is typically used as a classification algorithm, working off the assumption that similar points can be found near one another. Given the algorithm's simplicity and accuracy, it is one of the first classifiers that a new data scientist will learn. As new training samples are added, the algorithm adjusts to account for any new data since all training data is stored in memory. KNN only requires a k value and a distance metric, which is low when compared to other machine learning algorithms.

In FlaKat, we will be using Euclidean distance for calculating the neighbouring data points and explore the accuracy at a wide range of k [12].

4.5.2 Support Vector Machine

The support vector machine (SVM) [17] is the second option for categorizing the flaky tests. The objective of the SVM algorithm is to find a hyperplane in N-dimensional space(N — the number of features) that distinctly classifies the data points with the maximum margin. Maximizing the margin distance provides some reinforcement so that future data points can be classified with more confidence. The dataset of the flaky test does contain multiple labels and SVM doesn't support multiclass classification natively. The first adaptation is the One-to-One approach, which breaks down the multiclass problem into multiple binary classification problems. A binary classifier per each pair of classes. Another approach one can use is One-to-Rest. In that approach, the breakdown is set to a binary classifier per class.

The implementation by Scikit-learn will be used in FlaKat [14] and the decision function shape will be set to One-to-Rest. However, internally One-to-One is always used as a multi-class strategy to train models; a One-to-Rest matrix is only constructed from the

One-to-One matrix. Different kernel types will be explored to find out which yields the best accuracy.

4.5.3 Random Forest

The third option, Random forest [33], is a commonly-used machine learning algorithm, which combines the output of multiple decision trees to reach a single result. Its ease of use and flexibility have fueled its adoption, as it handles both classification and regression problems. The random forest algorithm is made up of a collection of decision trees, and each tree in the ensemble is comprised of a data sample drawn from a training set with replacement, called the bootstrap sample. Of that training sample, one-third of it is set aside as test data, known as the out-of-bag (oob) sample, which we'll come back to later. Another instance of randomness is then injected through feature bagging, adding more diversity to the dataset and reducing the correlation among decision trees. Depending on the type of problem, the determination of the prediction will vary. For a regression task, the individual decision trees will be averaged, and for a classification task, a majority vote—i.e. the most frequent categorical variable—will yield the predicted class. Finally, the oob sample is then used for cross-validation, finalizing that prediction.

Scikit-learn already implemented Random Forest classifier [13] and can be directly imported in FlaKat. Due to the high amount of tunable hyperparameters. Bayesian optimization will be applied to find the optimal configuration in categorizing flaky tests.

4.6 Bayesian Optimization

Bayesian Optimization [63] is added for optimal tuning of the Random Forest classifier. Bayesian Optimization provides a principled technique based on Bayes Theorem to direct a search of a global optimization problem that is efficient and effective. It works by building a probabilistic model of the objective function, called the surrogate function, that is then searched efficiently with an acquisition function before candidate samples are chosen for evaluation on the real objective function.

Such a method is supported by the open-source project Bayesian Optimization [1] and can be imported and used easily in FlaKat. The number of iterations is set based on the time limit of two hours and the objective function is trying to maximize the prediction accuracy. Optimization will take place twice: one with the objective function of F_1 score and the other using the newly proposed FDC.

4.7 Flakiness Detection Capacity

Due to the imbalance between flaky test categories and the drastic difference in their F_1 score, a new metric might be necessary to accurately measure the performance between different classifiers. Thus, thsi thesis propose the Flakiness Detection Capacity (FDC) to better compare the final results from the classifiers than using F_1 score. It is a new metric based on intrusion detection capacity derived from the field of information-theoretic analysis [28]. It quantifies the performance of a classifier and reflects the degree of correlation between the input flaky test vector and the output flaky category without knowing a reliable weighting policy between the flakiness categories. It is the ratio between the mutual information of vectorized embedding input and its category output to the entropy of input and can be calculated with the equation below.

$$\begin{aligned} FDC &= \frac{I(c_{in}; c_{out})}{H(c_{in})} \\ &= \frac{\sum_{c_{in}} \sum_{c_{out}} p(c_{in}, c_{out}) \, log \frac{p(c_{in}, c_{out})}{p(c_{in}) p(c_{out})}}{- \sum_{c_{in}} p(c_{in}) \, log \, p(c_{in})} \end{aligned} \tag{4.1}$$

, where c_{in} and c_{out} are the actual and predicted categories of flaky tests at the input and output of the machine learning classifier. Its value is the result of the division of the *mutual information* between Input Category (c_{in}) and Output Category (c_{out}) over the *entropy* H of Input Category (c_{in}). The value of I and H can be calculated with marginal probability mass function $p(c_{in})$ and $p(c_{out})$ and joint probability mass function $p(c_{in}, c_{out})$.

Applying F_1 score to the multi-class prediction of FlaKat could be misleading. In the work of FlaKat, the final F_1 score will be computed as the macro average with all categories of flaky tests sharing the same weight. The research has not agreed on a sound weighting policy for the categories defined in the IDoFT. FDC can avoid such issues and measure the performance of classifiers purely based on information theory without knowledge regarding the actual definition of categories.

4.8 Summary

The FlaKat framework is realized with the help of many existing tools. Git python library is a key component that enables automated parsing of the raw Java test case from their

corresponding Git repositories online. Three source code vectorization techniques, doc2vec, code2vec and tf-idf, are then applied to convert the newly parsed flaky tests into vector embedding which is then reduced to lower dimensionality by one of the PCA, LDA, Isomap, t-SNE and UMAP. The low dimensionality embedding with the best-preserved structure is split for training and testing machine learning classifier while sampling techniques, SMOTE and then Tomek Link, are applied to address the imbalance between different categories of flakiness. The performance of KNN, SVM and Random Forest will be explored to find their optimal configuration for categorizing flaky tests.

Chapter 5

Experiment and Evaluation

The evaluation process starts with the selection of a flaky test dataset. Three source code vectorization techniques, doc2vec, code2vec and tf-idf, are used to vectorize these flaky tests after parsing and preprocessing. Dimension reduction techniques, PCA, LDA, Isomap, t-SNE and UMAP, are applied to the vector embeddings which are analyzed both qualitatively and quantitatively for selecting the one that best preserves the local and global structure of flaky tests' vector embedding. Evaluation of the machine learning models, KNN, SVM and Random Forest, in the FlaKat framework, is based on two metrics: F_1 score and FDC. The first metric is the macro average of F_1 score for predicting the category of a flaky test. This metric is an indicator of how accurate the predictions are for machine learning algorithms. More detailed analysis on the F_1 score of individual categories will also be conducted. Then a novel metric defined in this study as flakiness detection capacity (FDC), is used as a secondary metric and its validity will also be proved. FDC is inspired by the calculation of intrusion detection capacity [28], which is the ratio between the mutual information of vectorized embedding inputs and its category output to the entropy of input. The goal of these metrics is to objectively measure how well an embedding reflects the category of flakiness. In the end, the final conclusion is made regarding which combination of code vectorization method, dimensionality reduction technique, and machine learning model yields optimal performance in flaky test categorization.

5.1 Data Set for Evaluation

A flaky test dataset that shows their flakiness category labels is required for training the machine learning models and evaluating the prediction. The International Dataset of Flaky

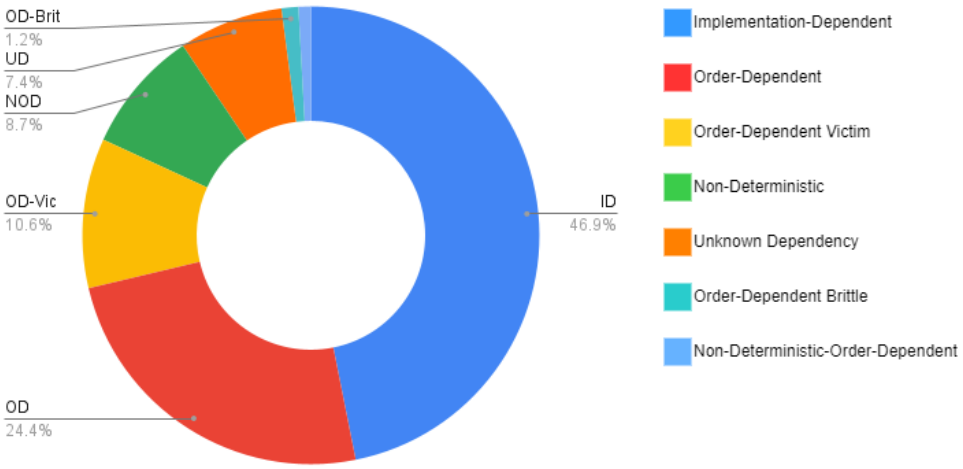


Figure 5.1: The imbalanced dataset before sampling from TDoFT

Test (IDoFT)[29, 46, 80] fulfills this requirement and provides thousands of real-world flaky tests that reside in numerous open-source projects. With the corresponding URL of all flaky tests, the framework can easily download the source code and analyze flaky Java test cases. The distribution of categories is shown in Fig. 5.1. In total, there are 1257 flaky tests from 108 open-source projects selected. Out of all tests, 589 of them are Implementation-Dependent (ID) manifest by [29], 307 of them are Order-Dependent (OD) labelled by [46], 133 and 15 of them are grouped into more specific Order-Dependent Victim (OD-Vic) and Brittle (OD-Brit) by [80]. In addition, there are 109 Non-Deterministic(NOD) flaky tests, and 93 have Unknown-Dependency (UD) that passes and fails in a test suite or isolation. There are also 11 Non-Deterministic Order-Dependent (NDOD) tests that fail non-deterministically but with significantly different failure rates in different orders. Non-Deterministic Order-Independent (NDOI) flaky tests have extremely low occurrence and often have other labels thus they are not selected for evaluation [48].

5.2 Dimensionality Reduction

Once parsed, the raw Java test cases are then converted into vector embeddings via doc2vec, code2vec or tf-idf with different dimension reduction techniques applied to them. As mentioned in the implementation details in Section 4.3, the techniques considered in FlaKat are PCA, LDA, Isomap, t-SNE, and UMAP. Unlike PCA and LDA, the Isomap, t-SNE and

UMAP reductions all have tunable hyperparameters that affect the result of the dimension reduction. The results from different numbers of neighbours to consider for Isomap, various perplexity for t-SNE, and numerous combinations of the number of neighbours and minimum distance between valid neighbours for UMAP, are collected as scatter plots and the ones with the cleanest clustering are selected for comparison with other reduction techniques both qualitatively and quantitatively. This procedure is based on the comparison performed between t-SNE and UMAP in [59].

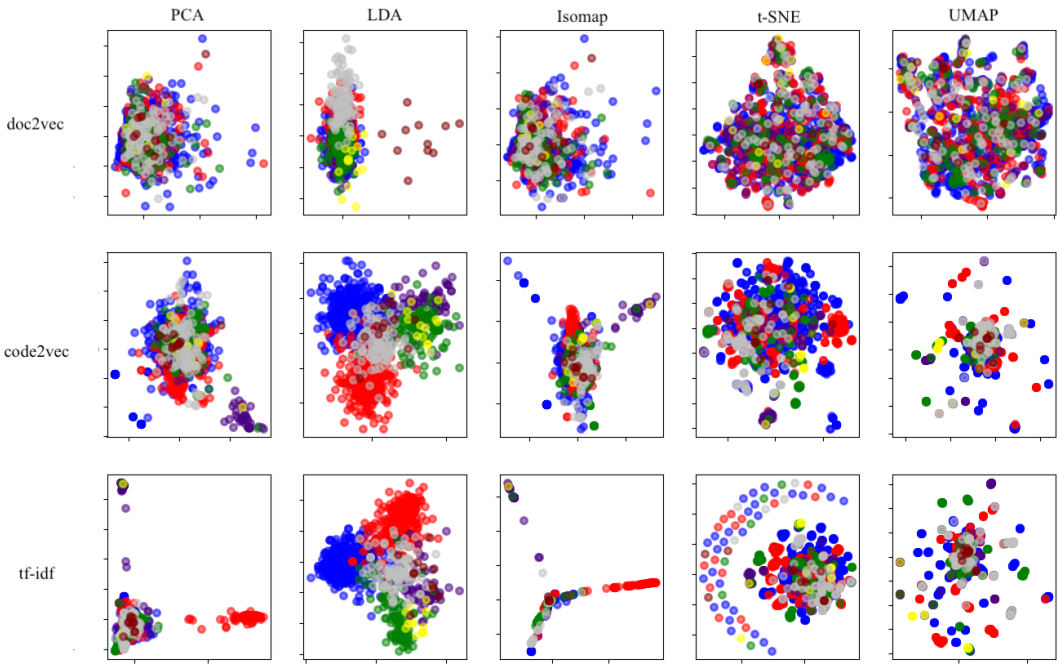


Figure 5.2: Summary of 2D projections from 3 vector embeddings by 5 dimensionality reduction techniques

5.2.1 Qualitative Analysis

The qualitative analysis of the dimensionality reduction techniques is first done by manual inspections of the visualized two-dimensional projection in Fig. 5.2. From left to right,

the scatter plots are the result of PCA, LDA, Isomap, t-SNE and UMAP respectively and from top to bottom, the plots are generated using doc2vec, code2vec and tf-idf. The data points from different flakiness categories in the reduced two-dimensional vector space are represented using the same colours as in Fig. 5.1. Though such an inspection does not yield accurate or precise conclusions, it can show a general idea of whether different categories of flaky tests are clustered closely together or not following dimensionality reduction.

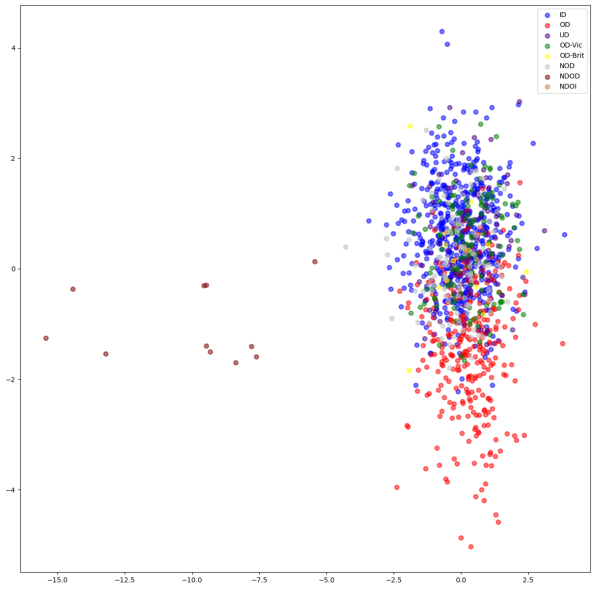
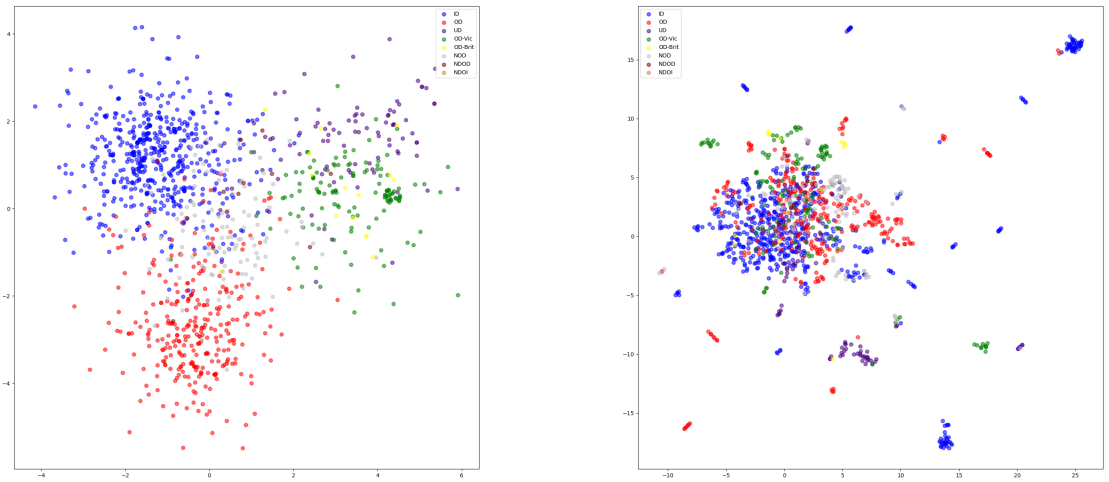


Figure 5.3: 2d projection of doc2vec embedding reduced by LDA

doc2vec

The PCA reduction technique applied on doc2vec embedding does not produce a promising result, as shown in its two-dimensional scatter plot. Most of the data points are located in a single cluster with a few ID (in blue) and OD (in red) data points randomly spread away. In the plot for LDA as shown in Fig. 5.3, NDOD (in brown) data points are clearly separated from others which are grouped in a huge cluster along the vertical axis with ND (in grey) data points located at the top, OD (in red) data points in the middle and the

rest in the bottom. Isomap's projection is quite similar to the one generated using PCA with no obvious clustering between the different categories except a few ID data points randomly located far from the large single cluster. The results from the manifold reduction methods t-SNE and UMAP are almost identical. Both of them seem worse compared to the previous techniques because all data points are evenly distributed in the plot, with different categories mixed up with one another.



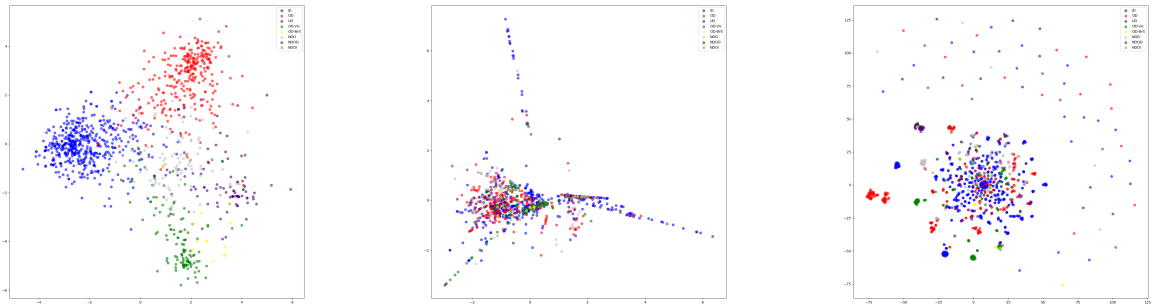
(a) LDA (b) UMAP

Figure 5.4: 2d projection of doc2vec embedding reduced by LDA and UMAP

code2vec

More clustering can be found in reduced embedding from code2vec. In the two-dimensional PCA projection, some UD (in violet) data points are clustered together while the data points from other categories overlap with each other. Compared to the reduced result from doc2vec, this result is marginally better. Fig. 5.4a shows the reduction by LDA which yields cleaner clustering for all categories except OD-Vic (in green) and OD-Brit (in yellow) data points, which are slightly mixed. The border between different clusters shows some overlapping as well. Such reduced projections can potentially result in better model performance. Data points in Isomap projections for code2vec embedding are less spread out

regardless of the value of neighbours compared to doc2vec's Isomap projection. However, the majority of data points overlap with each other except UD. Tuning perplexity and the number of iterations for t-SNE generate different reduction results, which all present small clusters of ID, OD, UD and OD-Vic while all remaining data points are almost evenly distributed in a big cluster at the center of projections. The results of UMAP for code2vec embedding are displayed in Fig. 5.4b not competitive. It shows similarly small clusters like t-SNE, but denser in all combinations of the number of neighbours and minimum distances. Moreover, it also possesses the same pattern with a huge cluster of flaky tests from different categories overlapping.



(a) LDA (b) Isomap (c) t-SNE

Figure 5.5: 2d projection of tf-idf embedding reduced by LdA, Isomap and t-SNE

tf-idf

As observed, in the two-dimensional PCA projection, some OD and UD data points are cleanly clustered while the rest are grouped into a single cluster with flaky tests from different categories. In contrast to PCA, LDA as shown in Fig. 5.5a yields a decent separation of clusters between all categories. Though there is some level of overlap at the boundaries of clusters, especially for NOD data points. Depending on the number of neighbours, Fig. 5.5b shows that Isomap yields different two-dimensional projections but the data points are usually densely located along several axes with no obvious clustering. This could be the result of the original distribution of data points in high dimensions being disconnected and having different densities causing difficult in unfolding. Applying t-SNE reduction also produces different outcomes depending on the perplexity setting and the number of iterations and the one with the cleanest separation is shown in Fig. 5.5c. In

most configurations, there are small clusters of ID, OD, UD, and OD-Vic. However, there is also another large cluster with mixed data points from all categories. UMAP, in theory, should outperform t-SNE but such property is unlikely to stay true when the constraint that a manifold exists for data points uniformly distributed in high-dimensional space is not met [59]. Here, similar to the UMAP's result for code2vec, the projection shows a huge cluster of flaky tests from different categories mixed together at the center. One interesting similarity is that the small clusters of single categories in both UMAP and t-SNE are indeed clusters of the same flaky test cases.

For all vector embeddings generated, especially by code2vec and tf-idf, the separation among data points from different flaky categories can be found from the manual inspection of data points' two-dimensional projections. Such clustering means the vectorization of raw Java test cases can preserve features related to flakiness's category making machine learning classification possible. However, overlapping data points can be observed near the border between clusters in most scatter plots. In general, the observations from some of the two-dimensional projections of vector embeddings show a clear separation of flaky test data points from different categories under certain settings. This is most notable in the plots from code2vec and tf-idf reduced by LDA.

5.2.2 Quantitative Analysis

K	**doc2vec**				
	PCA	**LDA**	**Isomap**	**t-SNE**	**UMAP**
2	0.20	**0.38**	0.20	0.22	0.22
5	0.17	**0.37**	0.17	0.19	0.19
10	0.15	**0.39**	0.16	0.17	0.20
20	0.15	**0.39**	0.15	0.17	0.18
50	0.14	**0.38**	0.12	0.15	0.15
100	0.15	**0.40**	0.13	0.15	0.16
200	0.13	**0.40**	0.13	0.13	0.15
500	0.10	**0.39**	0.14	0.12	0.12

Table 5.1: Quantitative Analysis with doc2vec using F_1 score

The quantitative analysis of dimensionality reduction is done by running the KNN algorithm on the reduced embeddings with various k values. When k (i.e. the number of neighbours to consider) is small, the results represent how well the local structure of a

cluster is preserved. On the other hand, when k is large, the results reflect how well data points structures are preserved globally. The minimum possible value for k is 2 due to the nature of the KNN algorithm and the largest value of k is set to 500. In addition, 10-fold cross-validation and sampling are applied for better estimation. The highest score for each embedding is highlighted for every k value.

doc2vec

In Table 5.1, the F_1 score of applying KNN on the reduced doc2vec embeddings is displayed with the k value ranging from 2 to 500. When k is small, embeddings reduced by LDA show much higher performance at 0.38 compared to others, and such superiority is maintained for all chosen k values. When k is large, the performance of LDA peaks at 0.40. The results from PCA, Isomap, t-SNE, and UMAP are similar with F_1 scores of 0.20 0.22 when k is 2, The F_1 scores decrease to 0.13 as k grows larger. Thus, the LDA reduction quantitatively yields the best-reduced vector embedding both locally and globally for doc2vec.

K	**code2vec**				
	PCA	**LDA**	**Isomap**	**t-SNE**	**UMAP**
2	0.36	**0.50**	0.43	0.54	0.44
5	0.34	**0.54**	0.39	0.51	0.42
10	0.34	**0.54**	0.37	0.48	0.42
20	0.31	**0.54**	0.35	0.45	0.41
50	0.30	**0.56**	0.32	0.40	0.40
100	0.27	**0.56**	0.28	0.37	0.37
200	0.27	**0.56**	0.24	0.33	0.32
500	0.26	**0.53**	0.28	0.23	0.25

Table 5.2: Quantitative Analysis with code2vec using F_1 score

code2vec

The performance of the KNN classifier using the vector embedding generated from code2vec is shown in Table 5.2 measured by F_1 score. LDA starts at 0.50 when k is at its minimum and increases to 0.56 as k reaches 200. It then declines to 0.53 when k is 500. Compared to the results of doc2vec, the LDA reduction for embedding generated by code2vec shows significantly higher performance than the other reduction methods. Additionally, the variance in F_1 score across different k values is also lower for LDA. This makes LDA

the best option for code2vec embedding among the considered reduction techniques. The UMAP reduction ranks second, marginally surpassing PCA, Isomap, and t-SNE in most situations.

K	tf-idf				
	PCA	**LDA**	**Isomap**	**t-SNE**	**UMAP**
2	0.40	0.53	0.45	**0.59**	0.49
5	0.37	0.54	0.43	**0.56**	0.44
10	0.36	**0.55**	0.38	0.52	0.44
20	0.34	**0.56**	0.35	0.49	0.41
50	0.33	**0.59**	0.33	0.41	0.36
100	0.30	**0.59**	0.27	0.32	0.29
200	0.30	**0.60**	0.22	0.30	0.23
500	0.23	**0.59**	0.16	0.23	0.20

Table 5.3: Quantitative Analysis with tf-idf using F_1 score

tf-idf

The outcome of tf-idf embedding is similar to that of code2vec as shown in Table 5.3. When k is at 2, the F_1 score from LDA is at 0.53, which is lower compared to 0.59 obtained for t-SNE. However, the gap between the LDA and t-SNE decreases as k increases, and F_1 score from LDA surpasses t-SNE once k becomes larger than 10. Although LDA is not the optimal technique for all values of k, its F_1 score is only sub-optimal by a small margin when k is small, while it yields significantly higher F_1 scores when k is large; thus, we conclude LDA to be the best reduction technique for tf-idf.

Based on the observations of both qualitative and quantitative analysis, the best dimensionality reduction technique of all three vector embedding methods is LDA. Qualitatively, it always demonstrates clearer clustering in the two-dimensional projection scatter plot compared to other techniques. Quantitatively, it also shows high performance measured in F_1 score when running KNN with the number of neighbours k between 2 to 500.

5.3 Imbalance and Sampling

The imbalance between the categories of different flaky tests could lead to unsatisfactory classification results; therefore, the sampling techniques mentioned in Phase IV are used to

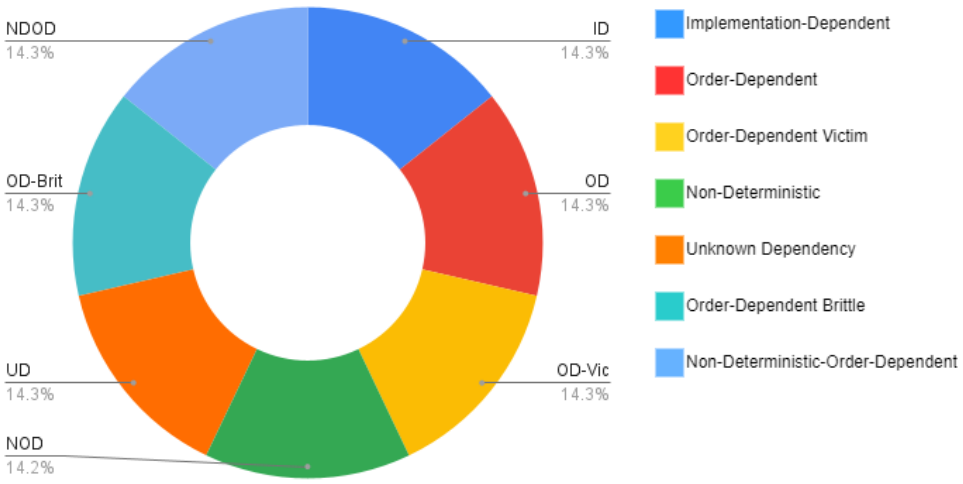


Figure 5.6: The balanced dataset after applying SMOTE and Tomek Link

address this problem. Synthetic data points for minority categories are created via SMOTE. On the other hand, TomekLinks is used to remove the bordering majority category data points for better separation between clusters. This process is applied to the training dataset during 10-fold cross-validation so there would be roughly 500 data points from each category and their distribution is as shown in Fig. 5.6. In the rare case that a minority class does not exist in the training set, another round of shuffle and split will be enforced until the lack of representation is resolved. Some flaky tests have multiple labels and they are dropped to maintain the simplicity of prediction. NDOI flaky tests have extremely low occurrence and often have other labels thus they are not selected for evaluation.

5.4 Prediction Accuracy for KNN and SVM

With the knowledge that vector embeddings of flaky tests are most well-preserved using LDA dimensionality reduction, FlaKat proceeds to the next stage where different machine learning models are evaluated to process the reduced dimensional vector representation of flaky tests and make classification of their categories. The performance of classification will be measured in two metrics. The first one is the F_1 score, which is a combination of precision and recall. The other is flakiness detection capacity (FDC), which is derived based on the work in [28] and can indicate the performance of models from an information-theoretic point of view. For both metrics, the experiment results are obtained as the average

of 10-fold cross-validation.

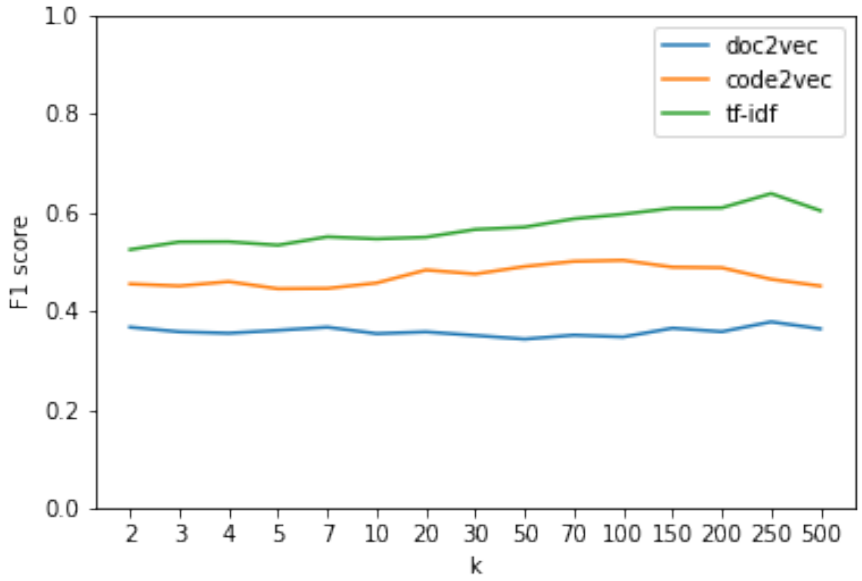


Figure 5.7: F_1 score of KNN classifier with different values for number of neighbours

The results from the KNN classifier for all 3 vector embeddings are shown in Fig. 5.7. It is palpable that predictions based on the embedding generated by doc2vec do not have high performance overall. As the value of k increases, its F_1 score remains around 0.37 regardless of k values. Both code2vec and tf-idf show improved performance compare to doc2vec, with the latter slightly better than the former across different values of k. When k is at its minimum value of 2, their F_1 scores are at 0.45 and 0.52 respectively. The code2vec approach reaches 0.50 when k reaches 70 and 100 but cannot maintains this performance for higher k values.As k increases, KNN prediction performance using tf-idf also increase until it reaches 500. The highest performance for tf-idf is 0.63, and it occurs when k is at 250. Overall, vector embeddings generated by tf-idf achieve the highest F_1 score for KNN.

The performance of different types of kernels, linear, polynomial, radial basis, and sigmoid, were explored for the SVM classifier in combination with the three embeddings. Each kernel type was also paired with the regularization parameter ranging from 0.1 to 100. The highest measured F_1 score is plotted in Fig. 5.8. Other tunable parameters use the default values from the scikit-learn package. The rank of F_1 scores among doc2vec, code2vec, and tf-idf is the same as the rank from KNN. Here, tf-idf also produced the

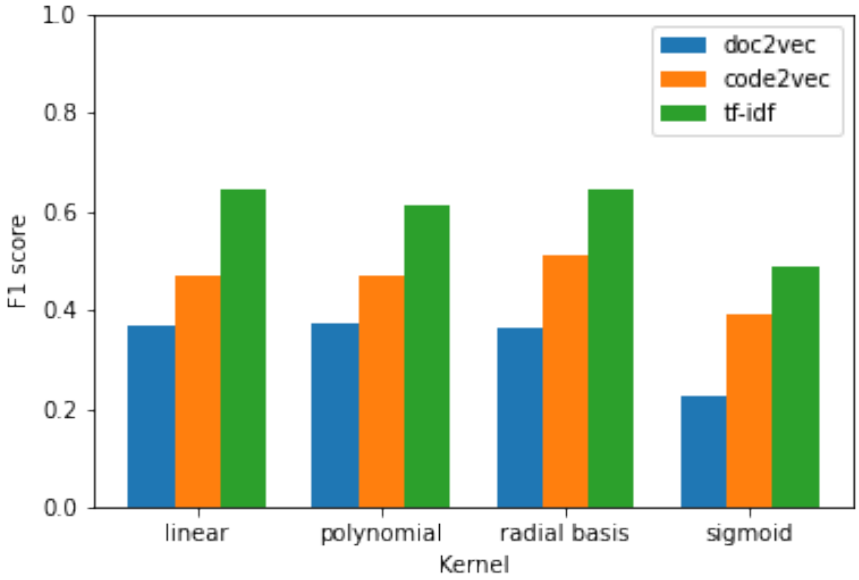


Figure 5.8: F_1 score of SVM classifier with different types of kernel

highest performance and doc2vec produced the lowest. For all 3 embeddings, their F_1 scores were around 0.36, 0.48, and 0.63 respectively across the different kernel types. In addition, their performance deteriorates with the sigmoid kernel and drops to 0.22, 0.39, and 0.48 respectively. By slight superiority, the highest F_1 score comes from tf-idf with linear and radial basis function kernel at 0.64.

5.5 Tuning Hyperparameters for Random Forest

5.5.1 Taguchi Orthogonal Array

Random Forest classifier is a powerful tool for data classification and regression. However, the number of tunable hyperparameters is much more than KNN and SVM. Finding the optimal configuration thus becomes a challenging task. To gain insight regarding the impact of each parameter on the categorization of flakiness, Taguchi Orthogonal array is implemented and applied [91].

The Taguchi method involves reducing the variation in a process through a robust

Feature	Config 1	Config 2	Config 3
n_estimators	50	100	150
criterion	gini	entropy	log_loss
max_depth	5	10	20
min_sample_split	20	40	60
min_sample_leaf	20	40	60
min_weight_fraction_leaf	0	0.01	0.02
max_features	sqrt	log2	None
min_impurity_decrease	0	0.001	0.01
ccp_alpha	0	0.01	0.03
max_leaf_nodes	None	50	100

Table 5.4: The 3 configurations for each hyperparameter that participating in the formation of Taguchi L27 Orthogonal Array

design of experiments. The overall objective of the method is to produce high-quality products at a low cost to the manufacturer. The Taguchi method was developed by Dr. Genichi Taguchi of Japan who maintained that variation. Taguchi developed a method for designing experiments to investigate how different parameters affect the mean and variance of a process performance characteristic that defines how well the process is functioning. The experimental design proposed by Taguchi involves using orthogonal arrays to organize the parameters affecting the process and the levels at which they should be varied. Instead of having to test all possible combinations like the factorial design, the Taguchi method tests pairs of combinations. This allows for the collection of the necessary data to determine which factors most affect product quality with a minimum amount of experimentation, thus saving time and resources.

The Taguchi method is best used when there is an intermediate number of variables (3 to 50), few interactions between variables, and when only a few variables contribute significantly which is suitable for the Ramdon Forest classifier supported by Scikit-learn library. There are 10 features that affect the performance of prediction and an L27 array is needed. We selected 3 values for each feature as shown in Table 5.4 and the experiment will be carried out results will be recorded for all 27 designated combinations including the average F_1 score and the variance of the F_1 score.

The prediction will be carried out 10 times which yields the result for variance and SN ratio. Once these SN ratio values are calculated for each factor and level, they are tabulated as shown below in Table 5.5 with the first 3 rows being the SN ratio and the last row being the maximum delta. The larger the delta is for a parameter, the larger the effect

n_estimators	criterion	max_depth	min_split	min_leaf
98.63402642	89.56220011	97.69543078	97.53378856	92.4371904
90.79673571	99.7660669	93.18646432	96.44596624	94.73134766
95.77125482	95.87374995	94.32012185	91.22226214	98.0334789
7.837290717	10.20386679	4.508966455	6.311526421	5.596288498
min_weight_frac	**max_features**	**min_impurity**	**ccp_alpha**	**max_leaf**
95.715144	93.72380519	93.70318004	96.90918742	93.86290561
93.63355033	95.05205375	94.30426889	98.17731409	95.98762827
95.85332262	96.51622772	97.29047466	94.0678465	95.35148307
2.219772296	2.792422534	3.587294624	4.109467588	2.124722654

Table 5.5: The scale of effect by different hypermeters on flaky test categorization

the variable has on the process. This is because the same change in signal causes a larger effect on the output variable being measured. Thus we can conclude The most impactful parameter on the effectiveness of Random Forest classifies in predicting the categories of flakiness is the criterion for splitting, the number of estimators which represents the number of the decision trees in the forest and the minimum sample size allowed for further splitting.

Despite the insight, finding the exact global optimal configuration still requires more effort.

5.5.2 Bayesian Optimization

Tuning the Random Forest classifier for optimal results is more challenging compared to KNN and SVM due to the high number of hyperparameters. Bayesian Optimization mentioned in Section 4.6 is incorporated into FlaKat with some additional modifications to address this issue. The objective function that needs to be maximized is the macro average F_1 score of flaky test prediction using Random Forest classifier on the embedding reduced by LDA through 5-fold cross-validation. All parameters that impact prediction performance for the Random Forest classifier are chosen for tuning. Their bounded parameter space regions are listed in the second column of Table 5.6. Bootstrap-related parameters are not included because bootstrap is disabled so that the whole dataset is used to build each tree. The rest of Table 5.6 displays the optimal configurations after 500 iterations. The macro average F_1 score from Random Forest using these settings are 0.35, 0.50 and 0.67 for doc2vec, code2vec, and tf-idf respectively.

Parameter	Bound	doc2vec	code2vec	tf-idf
max_depth	1-200	116	30	53
min_impurity_decrease	0-0.5	0.008	0.004	0.020
min_samples_leaf	1-200	34	41	70
min_samples_split	2-400	217	227	203
n_estimators	100-200	116	30	53
min_weight_fraction_leaf	0-0.05	0.042	0.002	0.005
max_leaf_nodes	22-400	286	125	324
criterion	gini,entropy,log_loss	log_loss	log_loss	log_loss
F_1 scores	N/A	0.35	0.50	0.67

Table 5.6: The bound and optimal value for Random Forest classifier parameters found Bayesian Optimization

5.6 Accuracy for Individual Category

Looking into the F_1 scores of individual categories revealed more details regarding the effectiveness of flaky test categorization for each classifier on different embeddings. The results displayed in Table 5.7 are from the optimal configuration of each classifier measured in average F_1 score. The categories are ordered according to their percentage in the original dataset before sampling. For doc2vec embedding, NDOD flaky tests obtain F_1 score of 1 for all KNN, SVM, and Random Forest classifiers but its overall averages are still much lower than code2vec and tf-idf. The score is consistent and reproducible but is likely caused by an insufficient amount of NDOD tests before over-sampling. There are more similarities between results from code2vec and tf-idf embeddings. ID and OD typed flaky tests can be predicted with high F_1 scores for code2vec embedding (0.90, 0.88, and 0.87) and tf-idf embedding (0.94, 0.93, and 0.93). OD-Brit and NDOD flaky tests show the lowest performance compared to all other categories. For code2vec embedding, their F_1 scores are around 0.15. For tf-idf embedding, their F_1 scores are around 0.38. These results are also aligned with the original distribution of the flaky tests in the dataset before sampling. ID and OD are the most common type of flaky tests and show clear clusters in the 2d projection while OD-Brit and NDOD are the rarest and mix with other flaky tests from OD-Vic, NOD, and UD.

Category	%	doc2vec			code2vec			tf-idf		
		KNN	**SVM**	**RF**	**KNN**	**SVM**	**RF**	**KNN**	**SVM**	**RF**
ID	46.8%	0.71	0.33	0.36	0.88	0.87	0.90	0.93	0.93	0.94
OD	24.4%	0.26	0.41	0.15	0.84	0.85	0.87	0.91	0.91	0.90
OD-Vic	10.5%	0.16	0.29	0.30	0.60	0.63	0.68	0.79	0.74	0.81
NOD	8.6%	0.28	0.24	0.61	0.34	0.34	0.45	0.46	0.48	0.54
UD	7.3%	0.12	0.19	0.19	0.49	0.49	0.56	0.60	0.57	0.56
OD-Brit	1.2%	0.08	0.12	0.13	0.19	0.20	0.35	0.47	0.52	0.50
NDOD	0.8%	1.0	1.0	1.0	0.14	0.15	0.21	0.33	0.38	0.42
Average	N/A	**0.37**	**0.37**	**0.39**	**0.50**	**0.51**	**0.59**	**0.63**	**0.64**	**0.67**

Table 5.7: The F_1 scores of the specific category of flaky tests using the optimal configurations for classifiers using embeddings generated by doc2vec, code2vec and tf-idf

5.7 Flakiness Detection Capacity

Inspired by intrusion detection capacity, Flakiness Detection Capacity (FDC) is introduced to quantify the performance of the classifier in predicting the categories of flaky tests from the information theory point of view. Before applying the FDC to the prediction result, its legitimacy and usefulness must be demonstrated. To prove one metric strictly superior to another is difficult and unlikely. Showing FDC is relatively more consistent and discriminant than F_1 score is sufficient to highlight the value of FDC using the same method as in [52].

# of reshuffle	5	10	20	50
Avg. C	0.77	0.77	0.79	0.80
Avg. D	1.63	2.17	1.97	1.86

Table 5.8: The average Consistency index and Discriminancy index of FDC compared against F_1 score

5.7.1 Consistency against F_1 score

The relative consistency index defined in [52] stated that for two measures f, g on domain Ψ which contains numerous sets of flaky tests, let $R = \{(a,b)|a,b \in \Psi, f(a) > f(b), g(a) > g(b)\}, S = \{(a,b)|a,b \in \Psi, f(a) > f(b), g(a) < g(b)\}$ the degree of consistency of f and g is C ($0 < C < 1$).

$$C = \frac{|R|}{|R| + |S|} \tag{5.1}$$

Given the definition, FDC is more consistent than F_1 score when C is larger than 0.5. It means among all combinations of set a and set b, the occurrence of FDC and F_1 scores agree with each other, $FDC(a) > FDC(b)$ AND $F_1score(a) > F_1score(b)$, is more frequent than the occurrence of them disagree.

5.7.2 Discriminancy against F_1 score

The relative discriminancy index defined in [52] stated that for two measures f, g on domain Ψ, let $P = \{(a,b)|a,b \in \Psi, f(a) \neq f(b), g(a) = g(b)\}, Q = \{(a,b)|a,b \in \Psi, g(a) = g(b), f(a) \neq f(b)\}$ the degree of discriminancy of f and g is D ($D > 0$), where

$$D = \frac{|P|}{|Q|} \tag{5.2}$$

Given the definition, FDC is more discriminant than F_1 score when D is larger than 1. It means among all combinations of set a and set b, the occurrence of FDC produces different results while F_1 score produces the same result, $FDC(a) \neq FDC(b)$ AND $F_1score(a) = F_1score(b)$, is more frequent than the occurrence of reverse.

5.7.3 Result Analysis

To calculate consistency index C with a decently large size of samples using Eq. 5.1, KNN, SVM, and Random Forest classifiers are trained and tested using the balanced IDoFT dataset with 5-fold splits and shuffling. The result of the FDC and F_1 score is then recorded and compared in Table 5.8. The process is repeated 5, 10, 20, and 50 times for consistency. The value of C remains stable at 0.79, 0.79, and 0.77 for the KNN, SVM, and Random Forest classifier, which is higher than the required 0.5 as stated in the definition of

the consistency index. A similar procedure is applied to calculating discriminancy index D from Eq. 5.2. For KNN, SVM, and Random Forest classifiers, their corresponding D values are 1.96, 1.93, 1.86 which are all higher than 1; therefore, FDC is also more discriminant than F_1 score.

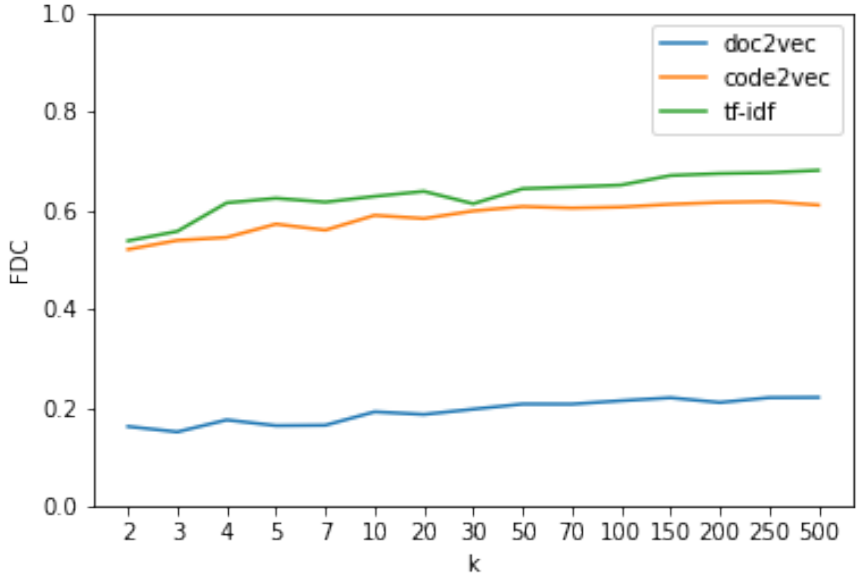


Figure 5.9: FDC of KNN classifier with different values for number of neighbours

The performance measured in FDC for KNN is displayed in Fig. 5.9. The ranking of the three vector embeddings is the same as the ranking measured in F_1 score. However, it does bring more insight regarding the impact of changes to the value of k. Unlike the F_1 score, FDC for doc2vec is significantly lower than the other two. The gap between cod2vec and tf-idf is also smaller. FDC of doc2vec starts at 0.16 (when k is small) and then gradually reaches 0.22 as k increases to 500. A similar trend appears in both code2vec and tf-idf results, where their FDCs are 0.52 and 0.53 respectively when k equals 2, and 0.61 and 0.68 respectively when k equals 500. For all k values, tf-idf embeddings produced a higher FDC value than the others.

In Fig. 5.10, the FDC measurements of SVM on the embeddings are almost identical to their F_1 score plot. Regardless of kernel type, the output from tf-idf is always more correlated to its vector embedding compared to code2vec. The optimal kernel type under FDC is the polynomial kernel with a score of 0.68. Linear and radial basis function kernel

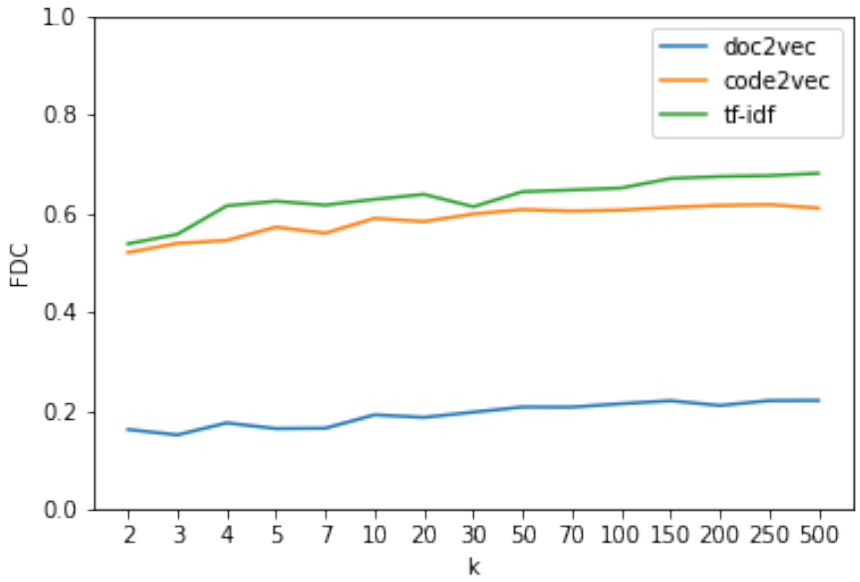


Figure 5.10: FDC of SVM classifier with different types of kernel

rank second together with slightly lower FDCs at 0.67. This observation is different from the earlier scenarios where performance is measured in F_1 score. Furthermore, doc2vec vector embeddings do not yield promising results.

The result of Bayesian Optimization using FDC as the objective function is shown in Table 5.9. The other choice of tuned parameters and their bounded region remain unchanged. Compared to the optimal configuration collected earlier, the classifier using doc2vec embedding obtained a higher minimum impurity required for splitting nodes and a lower maximum number of leaf nodes. This indicates the splits are less common and the classifier tuned in F_1 score likely suffers over-fitting. Code2vec embedding yielded the opposite outcome, as the max depth of the tree and max number of leaf nodes increased while the minimum sample leaf size and minimum sample required for splitting decreased. The overall splitting is more fine-grained compared to the previous experiment. The difference between the results from tf-idf embedding is relatively smaller. The maximum depth decreased while the minimum impurity required for splitting also decreased. Their FDCs are 0.22, 0.65, and 0.70 respectively, which is consistent with the results using F_1 score as the objective function.

Parameter	doc2vec	code2vec	tf-idf
max_depth	142	140	23
min_impurity_decrease	0.018	0.073	0.005
min_samples_leaf	66	16	105
min_samples_split	47	76	66
n_estimators	113	149	104
min_weight_fraction_leaf	0.039	0.033	0.002
max_leaf_nodes	148	226	338
criterion	entropy	log loss	gini
FDC	0.22	0.65	0.70

Table 5.9: The optimal value for Random Forest classifier parameters found Bayesian Optimization with FDC as objective function

5.8 Summary

The evaluation provides useful insight regarding the effectiveness of various components in FlaKat on flaky test categorization. The observations from some of the two-dimensional projections of vector embeddings show a clear separation of flaky test data points from different categories under certain settings. This is most notable in the plots from code2vec and tf-idf reduced by LDA. The reduced embeddings generated by LDA best preserves the local and global structure according to the performance of the KNN algorithm measured in F_1 score. Compared to doc2vec and code2vec, predictions made based on tf-idf embedding usually yield better results. The Random Forest classifier with parameters tuned after Bayesian Optimization obtains the highest F_1 score of 0.67 while SVM with linear kernel and regularization value of 0.1 scored 0.64 and KNN with k set to 250 scored 0.63. FDC is a better metric for evaluating flaky test categorization due to its higher consistency and discriminancy compared to F_1 score. Among all combinations of vector embeddings and classifiers, the highest FDC was obtained from tf-idf embedding and Random Forest classifier with parameters tuned by Bayesian Optimization.

Chapter 6

Conclusion

This research presents the framework, named FlaKat, for efficient and static flaky test categorization. First, the motivation behind building a novel flaky test categorization framework using the machine-learning approach is presented. Then, the workflow and details of the implementation of the framework with doc2vec, code2vec, and tf-idf source code representation are illustrated.

The final evaluation is done with real-world data from the IDoFT dataset via two metrics F_1 score and FDC. The results illustrate that both code2vec and tf-idf embeddings can closely reflect the flakiness category of test cases and help machine learning classifiers yield accurate predictions on certain categories of flakiness. To the best knowledge of the author, this is the first static flaky test categorization framework based on machine learning, which sets the foundation for future research in the software testing community.

6.1 Threats to Validity

There are several threats to the validity of this study. Some follow the design decisions while others follow the nature of the data used.

External: Size and labels of the dataset. Though the findings from the evaluations are supported by carefully formulated experiments, better results are possible if a larger dataset is available with more precise labels that are closely related to the content of the test case instead of its behaviour. One phenomenon observed in the dimensionality reduction of both tf-idf and code2vec vector representations is that there is a big cluster consisting of all categories of flaky tests for the two-dimensional projection from t-SNE and UMAP. This

could imply that the current categorization defined by the behaviour of test cases could be further explored. That is, some tests across different categories might have similar root causes or tests may share the same cause but be grouped into different categories. Despite the sampling techniques applied, the dominant (most commonly appearing) categories tend to obtain higher F_1 scores compared to the minorities (least commonly appearing) categories.

Internal: Overfitting. Some reduced two-dimensional projections do not provide a meaningful result, especially ones from doc2vec, even though their weighted F_1 score is relatively high. The reason is that there is no clear separation between categories in a huge cluster, and the optimal prediction configuration happened when the cleanly separated clusters are correctly predicted and the mixed overlapping clusters are rigorously split. More effort is required to handle the overlapping data points.

6.2 Research Contributions

The realization of the framework, FlaKat, is the primary contribution to the research community in the domain of flaky tests. It is inspired by earlier works in vocabulary-based machine learning frameworks and taken one step further. Throughout the development of FlaKat, several source code vectorization algorithms are applied and evaluated to determine which one best reflects the characteristics of flakiness. Several dimensionality reductions are used and compared both qualitatively and quantitatively. The effectiveness of machine learning models is also tuned and analyzed for flaky test categorization. The results show that different categories of flaky tests maintained their difference in vector space after been converted which lead to the success of FlaKat. The ID and OD flaky tests contribute up to 75% of total flaky tests in the data set and their F_1 scores achieve 0.94 and 0.90 respectively.

The value of Flakiness Detection Capacity is also demonstrated in this research. Compare to F_1 score, the newly proposed metric evaluates the flaky test categorization using information theory. FDC is applied FlaKat and its relative consistency index and discriminancy index are both higher than F_1 score which proves its superiority in flaky test categorization.

6.3 Future Directions

Several aspects of this work can be expanded upon and further explored. One direction worth investigating is the performance of the framework when the vector embedding is reduced to more than just two dimensions. There is a possibility that code2vec embedding produces better results than tf-idf when the reduced dimensionality is set to 5 or 10.

The number of features during the code vectorization can also be adjusted and might produce different outcomes. Currently, the vector embedding generated from doc2vec, code2vec and tf-idf has 384 features that come from the default settings of code2vec [3]. The effect of changing the number of features will spread to all other components in the framework and the result might change immensely.

Code clone detection tools, such as StoneDetector [5] and CCLearner [51], also have the potential for detecting flaky tests according to their category. This research has proved the characteristics of flaky tests are preserved in the vector embeddings generated by source code representation techniques. It is highly likely that tests from the same category share similarities in code tokens or syntax, which makes flakiness categorization achievable via clone detection with great efficiency.

Furthermore, an extension of the existing framework is to add a flaky test repair recommendation after successfully predicting the category of a test case. Some studies have been conducted in providing fixes for specific types of flaky tests in certain environments. A combined pipeline between these works and the labels from flakiness categorization output is the next step in the realization of such an automated repair tool.

References

[1] Bayesian optimization in python. https://github.com/fmfn/BayesianOptimization.

[2] Rui Abreu, Peter Zoeteweij, Rob Golsteijn, and Arjan J. C. van Gemund. A practical evaluation of spectrum-based fault localization. *J. Syst. Softw.*, 82(11):1780–1792, 2009.

[3] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. Code2vec: Learning distributed representations of code. *ACM Program. Lang.*, 3(POPL), 2019.

[4] Abdulrahman Alshammari, Christopher Morris, Michael Hilton, and Jonathan Bell. Flakeflagger: Predicting flakiness without rerunning tests. In *the International Conference on Software Engineering (ICSE)*, pages 1572–1584, 2021.

[5] Wolfram Amme, Thomas S. Heinze, and André Schäfer. You look so different: Finding structural clones and subclones in java source code. In *International Conference on Software Maintenance and Evolution (ICSME)*, pages 70–80, 2021.

[6] Jonathan Bell and Gail Kaiser. Unit test virtualization with vmvm. In *Proceedings of the International Conference on Software Engineering*, page 550–561, 2014.

[7] Jonathan Bell, Gail Kaiser, Eric Melski, and Mohan Dattatreya. Efficient dependency detection for safe java test acceleration. In *Proceedings of Joint Meeting on Foundations of Software Engineering*, page 770–781, 2015.

[8] Jonathan Bell, Owolabi Legunsen, Michael Hilton, Lamyaa Eloussi, Tifany Yung, and Darko Marinov. Deflaker: Automatically detecting flaky tests. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 433–444, 2018.

[9] Lionel C. Briand, Yvan Labiche, and Siyuan He. Automating regression test selection based on uml designs. *Information and Software Technology*, 51(1):16–30, 2009.

[10] Jeanderson Candido, Luis Melo, and Marcelo d'Amorim. Test suite parallelization in open-source projects: A study on its usage and impact. In *Proceedings of the International Conference on Automated Software Engineering*, page 838–848, 2017.

[11] Nitesh V. Chawla, Kevin W. Bowyer, Lawrence O. Hall, and W. Philip Kegelmeyer. Smote: synthetic minority over-sampling technique. *Journal of artificial intelligence research*, 16:321–357, 2002.

[12] KNN Classifier. https://scikit-learn.org/stable/modules/generated/sklearn.neighbors.KNeighborsClassifier.html.

[13] Random Forest Classifier. https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.htm.

[14] SVM Classifier. https://scikit-learn.org/stable/modules/generated/sklearn.svm.SVC.html.

[15] Code2vec. https://github.com/tech-srl/code2vec.

[16] Henry Coles, Thomas Laurent, Christopher Henard, Mike Papadakis, and Anthony Ventresque. Pit: A practical mutation testing tool for java (demo). In *Proceedings of the International Symposium on Software Testing and Analysis*, page 449–452, 2016.

[17] Corinna Cortes and Vladimir Naumovich Vapnik. Support-vector networks. *Machine Learning*, 20:273–297, 2004.

[18] T. Cover and P. Hart. Nearest neighbor pattern classification. *IEEE Transactions on Information Theory*, 13(1):21–27, 1967.

[19] Doc2vec. https://radimrehurek.com/gensim/models/doc2vec.html.

[20] Saikat Dutta, August Shi, Rutvik Choudhary, Zhekun Zhang, Aryaman Jain, and Sasa Misailovic. Detecting flaky tests in probabilistic and machine learning applications. In *Proceedings of the ACM International Symposium on Software Testing and Analysis(SIGSOFT)*, page 211–224, 2020.

[21] Moritz Eck, Fabio Palomba, Marco Castelluccio, and Alberto Bacchelli. Understanding flaky tests: The developer's perspective. In *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, page 830–840, 2019.

[22] Sebastian Elbaum, Hui Nee Chin, Matthew B. Dwyer, and Jonathan Dokulil. Carving differential unit test cases from system test cases. In *Proceedings of the ACM International Symposium on Foundations of Software Engineering (SIGSOFT)*, page 253–264, 2006.

[23] Lamyaa Eloussi. *Determining flaky tests from test failures.* 2015.

[24] Mona Erfani Joorabchi, Mehdi Mirzaaghaei, and Ali Mesbah. Works for me! characterizing non-reproducible bug reports. In *Proceedings of the Working Conference on Mining Software Repositories*, page 62–71, 2014.

[25] Sakina Fatima, Taher A. Ghaleb, and Lionel Briand. Flakify: A black-box, language model-based predictor for flaky tests. *IEEE Transactions on Software Engineering*, pages 1–17, 2022.

[26] GitPython. https://gitpython.readthedocs.io/en/stable/intro.html.

[27] Martin Gruber, Stephan Lukasczyk, Florian Kroiß, and Gordon Fraser. An empirical study of flaky tests in python. In *the IEEE Conference on Software Testing, Verification and Validation (ICST)*, pages 148–158, 2021.

[28] Guofei Gu, Prahlad Fogla, David Dagon, Wenke Lee, and Boris Skorić. Measuring intrusion detection capability: An information-theoretic approach. In *Proceedings of the ACM Symposium on Information, Computer and Communications Security*, page 90–101, 2006.

[29] Alex Gyori, Ben Lambeth, August Shi, Owolabi Legunsen, and Darko Marinov. Nondex: A tool for detecting and debugging wrong assumptions on java api specifications. In *Proceedings of the International Symposium on Foundations of Software Engineering (SIGSOFT)*, page 993–997, 2016.

[30] Guillaume Haben, Sarra Habchi, Mike Papadakis, Maxime Cordy, and Yves Le Traon. A replication study on the usability of code vocabulary in predicting flaky tests. In *the International Conference on Mining Software Repositories (MSR)*, pages 219–229, 2021.

[31] Mary Jean Harrold, James A. Jones, Tongyu Li, Donglin Liang, Alessandro Orso, Maikel Pennings, Saurabh Sinha, S. Alexander Spoon, and Ashish Gujarathi. Regression test selection for java software. In *Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (SIGPLAN)*, page 312–326, 2001.

[32] Michael Hilton, Timothy Tunnell, Kai Huang, Darko Marinov, and Danny Dig. Usage, costs, and benefits of continuous integration in open-source projects. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, page 426–437, 2016.

[33] Tin Kam Ho. The random subspace method for constructing decision forests. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 20(8):832–844, 1998.

[34] Hwa-You Hsu and Alessandro Orso. Mints: A general framework and tool for supporting test-suite minimization. In *the IEEE 31st International Conference on Software Engineering*, pages 419–429, 2009.

[35] Isomap. https://scikit-learn.org/stable/modules/generated/sklearn.manifold.Isomap.html.

[36] Yue Jia and Mark Harman. An analysis and survey of the development of mutation testing. *IEEE Transactions on Software Engineering*, 37(5):649–678, 2011.

[37] Bo Jiang, Zhenyu Zhang, W. K. Chan, and T. H. Tse. Adaptive random test case prioritization. In *the IEEE/ACM International Conference on Automated Software Engineering*, pages 233–244, 2009.

[38] Ian T. Jolliffe and Jorge Cadima. Principal component analysis: a review and recent developments. 2016.

[39] James A. Jones and Mary Jean Harrold. Empirical evaluation of the tarantula automatic fault-localization technique. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, page 273–282, 2005.

[40] Karen Sparck Jones. A statistical interpretation of term specificity and its application in retrieval. *Journal of documentation*, 1972.

[41] Taesoo Kim, Ramesh Chandra, and Nickolai Zeldovich. Optimizing unit test execution in large software programs using dependency analysis. In *Proceedings of the Asia-Pacific Workshop on Systems*, 2013.

[42] Adriaan Labuschagne, Laura Inozemtseva, and Reid Holmes. Measuring the cost of regression testing in practice: A study of java projects using continuous integration. In *Proceedings of the Joint Meeting on Foundations of Software Engineering*, page 821–830, 2017.

[43] Francis J. Lacoste. Killing the gatekeeper: Introducing a continuous integration system. In *Agile Conference*, pages 387–392, 2009.

[44] Wing Lam, Patrice Godefroid, Suman Nath, Anirudh Santhiar, and Suresh Thummalapenta. Root causing flaky tests in a large-scale industrial setting. In *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis*, page 101–111, 2019.

[45] Wing Lam, Kıvanç Muşlu, Hitesh Sajnani, and Suresh Thummalapenta. A study on the lifecycle of flaky tests. In *Proceedings of the International Conference on Software Engineering*, page 1471–1482, 2020.

[46] Wing Lam, Reed Oei, August Shi, Darko Marinov, and Tao Xie. idflakies: A framework for detecting and partially classifying flaky tests. In *the IEEE Conference on Software Testing, Validation and Verification (ICST)*, pages 312–322, 2019.

[47] Wing Lam, August Shi, Reed Oei, Sai Zhang, Michael D. Ernst, and Tao Xie. Dependent-test-aware regression testing techniques. In *Proceedings of the International Symposium on Software Testing and Analysis (SIGSOFT)*, page 298–311, 2020.

[48] Wing Lam, Stefan Winter, Angello Astorga, Victoria Stodden, and Darko Marinov. Understanding reproducibility and characteristics of flaky tests through test reruns in java projects. In *the International Symposium on Software Reliability Engineering (ISSRE)*, pages 403–413, 2020.

[49] LDA. https://scikit-learn.org/stable/modules/generated/sklearn.discriminantanalysis.LinearDiscriminantAnalysis.htm.

[50] Quoc Le and Tomas Mikolov. Distributed representations of sentences and documents. In *International conference on machine learning*, pages 1188–1196. PMLR, 2014.

[51] Liuqing Li, He Feng, Wenjie Zhuang, Na Meng, and Barbara Ryder. Cclearner: A deep learning-based clone detection approach. In *the IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 249–260, 2017.

[52] Charles X. Ling, Jin Huang, and Harry Zhang. Auc: A statistically consistent and more discriminating measure than accuracy. In *Proceedings of the International Joint Conference on Artificial Intelligence*, page 519–524, 2003.

[53] Tomek Link. https://imbalanced-learn.org/stable/references/generated/imblearn.undersampling.TomekLinks.html.

[54] Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. Learning from mistakes: A comprehensive study on real world concurrency bug characteristics. *SIGOPS Oper. Syst. Rev.*, 42(2):329–339, 2008.

[55] Qingzhou Luo, Farah Hariri, Lamyaa Eloussi, and Darko Marinov. An empirical analysis of flaky tests. In *Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering*, page 643–653, 2014.

[56] Qingzhou Luo, Farah Hariri, Lamyaa Eloussi, and Darko Marinov. An empirical analysis of flaky tests. In *Proceedings of the International Symposium on Foundations of Software Engineering (SIGSOFT)*, page 643–653, 2014.

[57] Mateusz Machalica, Alex Samylkin, Meredith Porth, and Satish Chandra. Predictive test selection. In *the International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pages 91–100, 2019.

[58] Jean Malm, Adnan Causevic, Björn Lisper, and Sigrid Eldh. Automated analysis of flakiness-mitigating delays. In *Proceedings of the International Conference on Automation of Software Test*, page 81–84, 2020.

[59] Leland McInnes, John Healy, Nathaniel Saul, and Lukas Großberger. Umap: Uniform manifold approximation and projection. *Journal of Open Source Software*, 3(29):861, 2018.

[60] Atif Memon, Zebao Gao, Bao Nguyen, Sanjeev Dhanda, Eric Nickell, Rob Siemborski, and John Micco. Taming google-scale continuous testing. In *the International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*, pages 233–242, 2017.

[61] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*, 2013.

[62] Sasa Misailovic, Aleksandar Milicevic, Nemanja Petrovic, Sarfraz Khurshid, and Darko Marinov. Parallel test generation and execution with korat. In *Proceedings of the Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, page 135–144, 2007.

[63] Jonas Mockus. *Bayesian approach to global optimization: theory and applications*, volume 37. 2012.

[64] Mika V. Mäntylä, Foutse Khomh, Bram Adams, Emelie Engström, and Kai Petersen. On rapid releases and software testing. In *the IEEE International Conference on Software Maintenance*, pages 20–29, 2013.

[65] Armin Najafi, Peter C. Rigby, and Weiyi Shang. Bisecting commits and modeling commit risk during testing. In *Proceedings of the Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, page 279–289, 2019.

[66] Agastya Nanda, Senthil Mani, Saurabh Sinha, Mary Jean Harrold, and Alessandro Orso. Regression testing in the presence of non-code changes. In *the IEEE International Conference on Software Testing, Verification and Validation*, pages 21–30, 2011.

[67] Alessandro Orso, Nanjuan Shi, and Mary Jean Harrold. Scaling regression testing to large software systems. In *Proceedings of the ACM Twelfth International Symposium on Foundations of Software Engineering (SIGSOFT)*, page 241–251, 2004.

[68] Owain Parry, Gregory M. Kapfhammer, Michael Hilton, and Phil McMinn. A survey of flaky tests. *ACM Trans. Softw. Eng. Methodol.*, 31(1), 2021.

[69] Samad Paydar and Aidin Azamnouri. An experimental study on flakiness and fragility of randoop regression test suites. In *Fundamentals of Software Engineering: 8th International Conference*, page 111–126, 2019.

[70] PCA. https://scikit-learn.org/stable/modules/generated/sklearn.decomposition.PCA.html.

[71] Gustavo Pinto, Breno Miranda, Supun Dissanayake, Marcelo d'Amorim, Christoph Treude, and Antonia Bertolino. What is the vocabulary of flaky tests? In *Proceedings of the International Conference on Mining Software Repositories*, page 492–502, 2020.

[72] Md Tajmilur Rahman and Peter C. Rigby. The impact of failing, flaky, and high failure tests on the number of crash reports associated with firefox builds. In *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, page 857–862, 2018.

[73] Alan Romano, Zihe Song, Sampath Grandhi, Wei Yang, and Weihang Wang. An empirical analysis of ui-based flaky tests. In *Proceedings of the International Conference on Software Engineering*, page 1585–1597, 2021.

[74] G. Rothermel, R.H. Untch, Chengyun Chu, and M.J. Harrold. Prioritizing test cases for regression testing. *the IEEE Transactions on Software Engineering*, 27(10):929–948, 2001.

[75] Matthew J. Rummel, Gregory M. Kapfhammer, and Andrew Thall. Towards the prioritization of regression test suites with data flow information. In *Proceedings of the ACM Symposium on Applied Computing*, page 1499–1504, 2005.

[76] David Saff, Shay Artzi, Jeff H. Perkins, and Michael D. Ernst. Automatic test factoring for java. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, page 114–123, 2005.

[77] Tony Savor, Mitchell Douglas, Michael Gentili, Laurie Williams, Kent Beck, and Michael Stumm. Continuous deployment at facebook and oanda. In *the International Conference on Software Engineering Companion (ICSE-C)*, pages 21–30, 2016.

[78] August Shi, Jonathan Bell, and Darko Marinov. Mitigating the effects of flaky tests on mutation testing. In *Proceedings of the International Symposium on Software Testing and Analysis (SIGSOFT)*, page 112–122, 2019.

[79] August Shi, Alex Gyori, Owolabi Legunsen, and Darko Marinov. Detecting assumptions on deterministic implementations of non-deterministic specifications. In *the International Conference on Software Testing, Verification and Validation (ICST)*, pages 80–90, 2016.

[80] August Shi, Wing Lam, Reed Oei, Tao Xie, and Darko Marinov. Ifixflakies: A framework for automatically fixing order-dependent flaky tests. In *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, page 545–555, 2019.

[81] Denini Silva, Leopoldo Teixeira, and Marcelo d'Amorim. Shake it! detecting flaky tests caused by concurrency with shaker. In *the International Conference on Software Maintenance and Evolution (ICSME)*, pages 301–311, 2020.

[82] SMOTE. https://imbalanced-learn.org/stable/references/generated/imblearn.oversampling.SMOTE.html.

[83] Amitabh Srivastava and Jay Thiagarajan. Effectively prioritizing tests in development environment. In *Proceedings of the ACM International Symposium on Software Testing and Analysis (SIGSOFT)*, page 97–106, 2002.

[84] Friedrich Steimann, Marcus Frenkel, and Rui Abreu. Threats to the validity and value of empirical assessments of the accuracy of coverage-based fault locators. In *Proceedings of the International Symposium on Software Testing and Analysis*, page 314–324, 2013.

[85] t SNE. https://scikit-learn.org/stable/modules/generated/sklearn.manifold.TSNE.html.

[86] Lin Tan, Chen Liu, Zhenmin Li, Xuanhui Wang, Yuanyuan Zhou, and Chengxiang Zhai. Bug characteristics in open source software. *Empirical Softw. Engg.*, 19(6):1665–1705, 2014.

[87] Joshua B. Tenenbaum, Vin de Silva, and John C. Langford. A global geometric framework for nonlinear dimensionality reduction. *Science*, 290(5500):2319–2323, 2000.

[88] Tf-idf. https://scikit-learn.org/stable/modules/generated/sklearn.featureextraction.text.TfidfVectorizer.html.

[89] Alaa Tharwat, Tarek Gaber, Abdelhameed Ibrahim, and Aboul Ella Hassanien. Linear discriminant analysis: A detailed tutorial. *Ai Communications*, 30:169–190, 2017.

[90] Ivan Tomek. Two modifications of cnn. *IEEE Transactions on Systems, Man, and Cybernetics*, SMC-6(11):769–772, 1976.

[91] KWOK-LEUNG TSUI. An overview of taguchi method and newly developed statistical methods for robust design. *IIE Transactions*, 24(5):44–57, 1992.

[92] UMAP. https://umap-learn.readthedocs.io/en/latest/.

[93] Arash Vahabzadeh, Amin Milani Fard, and Ali Mesbah. An empirical study of bugs in test code. In *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 101–110, 2015.

[94] Laurens van der Maaten and Geoffrey Hinton. Viualizing data using t-sne. *Journal of Machine Learning Research*, 9:2579–2605, 2008.

[95] Béla Vancsics, Tamás Gergely, and Árpád Beszédes. Simulating the effect of test flakiness on fault localization effectiveness. In *IEEE Workshop on Validation, Analysis and Evolution of Software Tests (VST)*, pages 28–35, 2020.

[96] Roberto Verdecchia, Emilio Cruciani, Breno Miranda, and Antonia Bertolino. Know you neighbor: Fast static prediction of test flakiness. *IEEE Access*, 9:76119–76134, 2021.

[97] Frank Wilcoxon. *Individual Comparisons by Ranking Methods*, pages 196–202. 1992.

[98] W. Eric Wong, Vidroha Debroy, Ruizhi Gao, and Yihao Li. The dstar method for effective software fault localization. *IEEE Transactions on Reliability*, 63(1):290–308, 2014.

[99] Ming Wu, Fan Long, Xi Wang, Zhilei Xu, Haoxiang Lin, Xuezheng Liu, Zhenyu Guo, Huayang Guo, Lidong Zhou, and Zheng Zhang. Language-based replay via data flow cut. In *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering*, page 197–206, 2010.

[100] A. Zeller and R. Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering*, 28(2):183–200, 2002.

[101] Lingming Zhang, Darko Marinov, Lu Zhang, and Sarfraz Khurshid. Regression mutation testing. In *Proceedings of the International Symposium on Software Testing and Analysis*, page 331–341, 2012.

[102] Lingming Zhang, Lu Zhang, and Sarfraz Khurshid. Injecting mechanical faults to localize developer faults for evolving software. *SIGPLAN Not.*, 48(10):765–784, 2013.

[103] Peilun Zhang, Yanjie Jiang, Anjiang Wei, Victoria Stodden, Darko Marinov, and August Shi. Domain-specific fixes for flaky tests with wrong assumptions on underdetermined specifications. In *the International Conference on Software Engineering (ICSE)*, pages 50–61, 2021.

[104] Sai Zhang, Darioush Jalali, Jochen Wuttke, Kıvanç Muşlu, Wing Lam, Michael D. Ernst, and David Notkin. Empirically revisiting the test independence assumption. In *Proceedings of the International Symposium on Software Testing and Analysis*, page 385–396, 2014.

[105] Celal Ziftci and Diego 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)*, pages 736–745, 2020.