# FlakeFlagger: Predicting Flakiness Without Rerunning Tests

Abdulrahman Alshammari[1], Christopher Morris[2], Michael Hilton[2] and Jonathan Bell[3]
[1]George Mason University, Fairfax, VA, USA
[2]Carnegie Mellon University, Pittsburgh, PA, USA
[3]Northeastern University, Boston, MA, USA
aalsha2@gmu.edu, christom@andrew.cmu.edu, mhilton@cmu.edu, j.bell@northeastern.edu

***Abstract*—When developers make changes to their code, they typically run regression tests to detect if their recent changes (re)introduce any bugs. However, many tests are *flaky*, and their outcomes can change non-deterministically, failing without apparent cause. Flaky tests are a significant nuisance in the development process, since they make it more difficult for developers to trust the outcome of their tests, and hence, it is important to know which tests are flaky. The traditional approach to identify flaky tests is to rerun them multiple times: if a test is observed both passing and failing on the same code, it is definitely flaky. We 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. We propose FlakeFlagger, a novel approach that collects a set of features describing the behavior of each test, and then predicts tests that are likely to be flaky based on similar behavioral features. We found that FlakeFlagger correctly labeled 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 on our dataset of 23 projects with flaky tests, FlakeFlagger outperformed the prior approach (by F1 score) on 16 projects and tied on 4 projects. Our results indicate that this approach can be effective for identifying likely flaky tests prior to running time-consuming flaky test detectors.**

## I. Introduction

Regression testing is widely used in quality assurance to determine if recent changes to a codebase (re)introduce errors. When a test fails, developers typically expect that this failure represents a bug recently introduced. However, a growing and concerning trend is regression tests failures which are not in fact due to recent changes, but are instead *flaky tests*.

Flaky tests are non-deterministic tests which pass and fail when run on the exact same version of a codebase [1]. When tests alternate between passing and failing without any code changes, developers can be frustrated: flaky tests are challenging to debug, and a single failing test can halt release cycles. Recent academic and industrial studies have worked towards defining and reporting flaky tests [1]–[5], automatically detecting flaky tests [6], [7], and investigating the occurrence of flakiness [8]–[10]. In fact, flaky tests appear in most large-scale projects. Google has reported that 4.56% of test failures are caused by flaky tests [1], and Microsoft's Windows and Dynamics products report a similar 5% [5].

Flaky test detection is crucial because flakiness reduces tests' reliability and frustrates developers. When flaky tests only fail occasionally, these tests can be very hard to reproduce [1], and developers then spend a significant amount of time and resources trying to fix these flaky failures. Because non-deterministic failures are not necessarily a result of a regression in the code, the very nature of flakiness means developers have a very difficult time locating the source of the flakiness. Hence, automatically determining if a test is flaky or not has become a significant topic in recent software testing research [1], [5], [6], [9], [11]–[15].

Existing techniques for detecting flaky tests rely on rerunning tests: if we can witness two different outcomes (passing and failing) from the same test on the same version of the codebase, then surely that test is flaky. Even state-of-the-art techniques like NonDex [16] and iDFlakies [17] that inject additional non-determinism still rely on re-running each test case many times. However, rerunning tests can be quite expensive. Most prior academic studies that detected which tests were flaky have re-executed test suites a handful of times: 10 times [6], 16 times [7] or 100 times [17]. Developers have reported re-running suspected flaky tests up to 1,000 times to increase the likelihood of determining if a test is flaky [18].

In this paper, we describe an empirical study of flaky tests that we conducted by rerunning tests *10,000 times* each, which can provide guidance to developers and other researchers on how many times to re-run tests to detect flakiness. While rerunning tests 10,000 times may be impractical for developers to perform regularly (or even for researchers to perform regularly to gather datasets), we used the result of this experiment to construct a rich dataset of flaky tests. This dataset has many uses, for instance, to study how developers can better identify, repair and prevent flaky tests.

A promising alternative approach to detect flaky tests is to create a machine learning classifier that can distinguish between flaky and non-flaky tests [12], [19]. In such an approach, developers train a classifier using a known corpus of flaky and non-flaky tests, and then apply that classifier to a new codebase in order to detect new flaky tests. Alternatively, rather than re-run *all* tests in a corpus thousands of times in order to find flaky tests, developers or researchers could use

this classifier to determine which tests are *more likely* to be flaky, and focus computational resources on re-running those tests first. Unfortunately, we found that existing approaches are not suitable to this sort of use case: in our evaluation, a state-of-the-art flaky test classifier had promising recall (72%), but extremely low precision (11%).

Ideally, the features used by the classifier should be applicable to a broad range of projects, so that a model built on one set of projects (with known flaky tests) could be applied to a new project to find new flaky tests. A recent survey has found 23 factors that increase, decrease and otherwise affect the ability to identify flakiness in tests [8]. These factors include features such as the presence of test smells, hard coded values, the age of test cases, and more. Inspired by this work, we collect various features like these for each of the tests in our flaky test dataset and created FlakeFlagger, a tool to automatically predict if a test is flaky or not using these features.

We evaluated FlakeFlagger on our large dataset of flaky tests, and found it significantly more effective than the state-of-the-art flaky test classifier: On the 23 projects evaluated, FlakeFlagger outperformed the prior approach in 16 and underperformed in only 3 projects — where two of them have only three flaky tests in total. The four remaining projects have 0% F1 score in the FlakeFlagger and the state-of-the-art flaky test classifier evaluation. Section IV presents a complete per-project evaluation.

This paper makes the following main contributions:

- **Dataset:** A new dataset of flaky tests created by exhaustively executing 24 project's test suites 10,000 times each — orders of magnitude more than prior work [6], [7], [12]. We show that for many flaky tests, it is necessary to rerun them very many times to detect them.
- **New Approach and Tool:** FlakeFlagger, a new hybrid static/dynamic approach to collect behavioral features of tests and then predict whether tests are flaky or not. FlakeFlagger can be used to guide other flaky test detection tools towards those most likely to be flaky.
- **Evaluation:** We show that FlakeFlagger can predict whether a test is flaky or not with far fewer false positives than prior work (just 380 false positives, compared to 4,674 from prior work). We show that test execution time, test coverage of source code, test coverage of changed lines, and usage of third party libraries are effective predictors of flakiness. We do not find any tokens that commonly exist in flaky tests, and we do not find test smells to be a strong indicator of flakiness.

Our dataset, test running infrastructure, and FlakeFlagger itself are publicly released under a BSD license and can be found on GitHub [20] and in our permanently archived artifact [21].

## II. Empirical Study

The traditional approach to find flaky tests is to rerun each test many times: if a test outcome changes (without any change to the code), then the test is flaky. This definition captures both tests that typically pass (but infrequently fail), and those that typically fail (but infrequently pass). However, there is little guidance (from industry or academia) in terms of how many times to rerun each test: most prior work has considered rerunning 10 [6], 16 [7], 100 times [17] or in limited cases, 4,000 times [22]. But, if each flaky test is flaky for some non-deterministic reason, what confidence do we have that we will observe its outcome change within those reruns? If it is feasible to find most flaky tests after only rerunning them a handful of times, then perhaps this flaky test rerun problem is not such a big problem: rerunning tests only 3-5 times does take time, but it certainly takes less time than rerunning them 10,000 times. Since we don't know the underlying source of non-determinism that causes the flaky test to fail, a *separate* problem is: how much harder is it for a developer to reproduce that flaky failure *in a different environment*, where various uncontrolled factors might vary (e.g., OS, Java version, CPU, available memory, network speed, etc.). This problem is particularly relevant for developers who are tasked with debugging a flaky test on their local machines which failed on a build server. To motivate our work in detecting flaky tests, we conducted an empirical study, investigating these questions:

**MQ1:** How many flaky tests can be found by rerunning tests given different rerun budgets?

**MQ2:** How hard is it to reproduce a flaky test failure?

These questions are important not only for this work, but for any researcher interested in developing new techniques to help developers detect and debug flaky tests.

### A. Study Design

To answer these questions, we selected 24 projects that have been previously studied in the context of flaky tests [6], [7]. In comparison, the dataset of flaky tests produced by Bell et al.'s DeFlaker [6] was constructed by running tests once on a revision of a project, then only rerunning tests that were observed to fail (up to 15 times). Lam et al.'s iDFlakies dataset limited the number of reruns per-project to take no more than 56 hours per-project, resulting in typically under 100 reruns per-project, and often did not run all tests in every project [7]. While Lam et al. have also conducted evaluations with more test re-runs (4,000), these evaluations were targeted to only portions of test suites with known flaky tests [22]. While this approach can reduce the overall amount of computation time needed for an experiment (Lam et al. report this experiment consumed 90 days of computation time), focusing on rerunning only known flaky tests can bias a dataset to only include those flaky tests that had previously been found through fewer reruns. Instead, we ran each project's *entire* test suite 10,000 times, using over 5 years of computation time in total.

We executed this experiment by creating a queue of jobs, where each job represented a single execution of a project's test suite (running "mvn install"). After running each job, we archived all log files and then removed all temporary files and rebooted the machine, and then proceeded to run the next test suite in the queue. This approach was designed to provide reasonable isolation between test runs, and simulates how a real build server might compile and test a project.

TABLE I: Flaky tests detected by re-running test suites 10,000 times. We estimate the percentage of all flaky tests that would be detected if only 10, 100 or 1,000 reruns had been performed. Color bars are stacked bar charts showing the percentage of tests that failed with a given frequency. Columns *DeFlaker* and *IDFlakies* show the number of non-order dependent flaky tests found in total by those prior works, and the number of flaky tests shared by both datasets. Blank cells indicate that a different revision of the project was used due to historical compilation issues. Complete data and results available in our artifact [21].

| Project | Tests | Flaky by Reruns | DeFlaker [6] Shared | DeFlaker [6] Total | iDFlakies [7] Shared | iDFlakies [7] Total | % of all Flaky Tests Detectable at: 10 Reruns | 100 Reruns | 1,000 Reruns | Distribution of Failure Frequencies, as % of Tests Failing (0,10] (10, 100] (100, 1,000] (1,000, 10,000] runs of 10,000 |
|---|---|---|---|---|---|---|---|---|---|---|
| spring-boot | 2,128 | 163 | 0 | 5 | | | 71% | 71% | 77% | |
| hbase | 431 | 145 | 0 | 1 | | | 52% | 59% | 75% | |
| alluxio | 187 | 116 | 2 | 2 | | | 0% | 91% | 100% | |
| okhttp | 810 | 100 | | | | | 8% | 12% | 15% | |
| ambari | 324 | 52 | 1 | 1 | | | 0% | 2% | 94% | |
| hector | 142 | 33 | 1 | 1 | | | 3% | 3% | 100% | |
| activiti | 2,044 | 32 | | | | | 0% | 3% | 44% | |
| java-websocket | 145 | 23 | | | 22 | 52 | 0% | 26% | 87% | |
| wildfly | 1,238 | 23 | | | | | 0% | 0% | 4% | |
| httpcore | 712 | 22 | 1 | 1 | | | 0% | 9% | 9% | |
| logback | 842 | 22 | | | | | 5% | 9% | 41% | |
| incubator-dubbo | 2,177 | 19 | | | 5 | 12 | 5% | 11% | 26% | |
| http-request | 163 | 18 | | | | | 0% | 83% | 83% | |
| wro4j | 1,145 | 16 | 1 | 1 | | | 44% | 50% | 81% | |
| orbit | 86 | 7 | 0 | 1 | | | 14% | 43% | 86% | |
| undertow | 183 | 7 | 0 | 3 | | | 0% | 0% | 29% | |
| achilles | 1,317 | 4 | | | | | 0% | 25% | 75% | |
| elastic-job-lite | 558 | 3 | | | 1 | 6 | 0% | 0% | 0% | |
| zxing | 345 | 2 | 2 | 2 | | | 0% | 100% | 100% | |
| assertj-core | 6,267 | 1 | 1 | 1 | | | 0% | 100% | 100% | |
| commons-exec | 55 | 1 | | | | | 0% | 0% | 100% | |
| handlebars.java | 428 | 1 | | | | | 0% | 100% | 100% | |
| ninja | 306 | 1 | 1 | 1 | | | 0% | 100% | 100% | |
| jimfs | 212 | 0 | | | | | | | | No flaky tests observed |
| **Total** | 22,245 | 811 | 10 | 20 | 28 | 70 | 26% | 45% | 67% | |

We considered approaches to increase the likelihood of flaky test detection. For instance, some tests might be flaky because they assume that a standard library method provides deterministic behavior, even when it doesn't (e.g., an iterator over a set, which might return objects in the same order each time, or might not) [16]. Other flaky tests may be *order dependent* [23], which means that when they run in a different order than is expected, they can fail (be flaky). Rather than simply re-run such tests 10,000 times, one could rerun them in a different order each time, thereby increasing the likelihood of observing a flaky test [7]. However, there are far more root causes of flaky tests than only these two [1], and we did not want to bias our dataset to include more flaky tests of any single category than another. For instance: in their survey of developer-reported flaky tests, Luo et al. found that test order dependencies accounted for only 13% of the flaky tests that developers reported in issue trackers [1]. Instead, our dataset represents the flaky tests that would be observed in the course of normal development, when tests are run on a build server — with no artificial non-determinism.

We ran these experiments on Ubuntu 18.04 VMs running OpenJDK 1.8.0_242 with 4 CPUs and 8GB of RAM. Note that perhaps, we could also find more flaky tests by using a variety of platforms and computing devices: using different versions of Java and different operating systems might introduce more non-determinism overall, finding more flaky tests. However, again, our goal in this study is to find the flaky tests that a developer might observe in the normal course of development when running their test suite on a standard continuous integration build server, and *not* to inject artificial noise. Hence, each execution of each test occurs using a standard platform.

### B. Study Results

By running each test 10,000 times, we observed 811 flaky tests: about 3.6% of the total number of tests were flaky. The number of observed flaky tests varies from one project to another. We found 10 projects with less than 10 flaky tests, 4 of which have only one, and one with none. On the other hand, there are 4 projects which have more than 100 flaky tests. *Spring-boot* has 163 flaky tests, 20% of the total observed flaky tests. Table I summarizes these results.

**MQ1: Flaky failures by number of reruns**. While it is not possible to truly state what the probability is of a flaky test failing (since we don't know the hidden, uncontrolled condition that causes the failure), we estimate the probability that each flaky test would have been detected with fewer reruns by assuming that each test failure occurs independently. Overall, only roughly a quarter of all of the flaky tests that we found in 10,000 runs would have been found with 10 reruns, roughly half with 100 reruns and roughly two thirds with 1,000 reruns. We also visualize this distribution in Table I, categorizing each flaky test as failing either between 0 and 10 times, between 10 and 100 times, between 100 and 1,000 times, and finally over 1,000 times (out of the 10,000 runs). These sets are represented by colors as shown in Table I. The *red* bar, which refers to tests that flake less than or equal 10

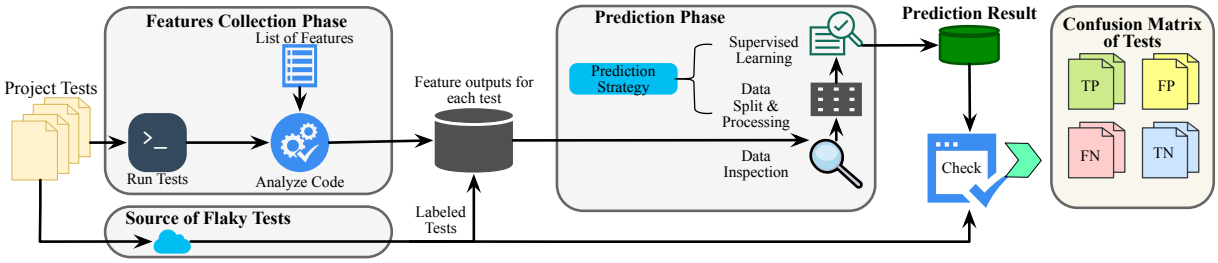


Fig. 1: Overview of FlakeFlagger's approach to predict likely flaky tests given a set of known flaky tests.

times, takes the majority in 8 projects. In general, we found more than 33% of total flaky tests fail in less than or equal 10 times, 22% of flaky tests fail more than 10 and less than or equal 100 out of 10,000. This suggests that researchers building flaky test datasets should consider as many reruns as possible in order to detect flaky tests: the chance of observing that a test is flaky might be quite slim indeed. Furthermore, we acknowledge that even after 10,000 re-runs, it is still possible that we have not detected every flaky test in this dataset. However, this threat is present in *any* flaky test dataset, and we believe that our empirical design is sufficient for our needs.

**MQ2: Reproducing Flaky Tests Failures.** While MQ1 provides insight into the difficulty of identifying flaky tests by re-running them on the same platform, this does not quite capture the challenge that a developer would face when reproducing those flaky tests in a different environment (e.g. their local machine). To simulate this activity, we compared the set of flaky tests identified from our 10,000 reruns with those detected by prior researchers on the same versions of the same projects, but in different environments. We specifically chose the projects and revisions of the projects that we studied in order to align with revisions of projects studied by either DeFlaker [6] or iDFlakies [7]. The columns *DeFlaker* and *iDFlakies* in Table I show the total number of flaky tests that that paper reported on that version of that project, along with the number of those tests that were also found to be flaky based on our reruns. A blank entry indicates that the revision of that project that we executed did not have any flaky tests reported by the prior work. The DeFlaker dataset contains flaky tests from many revisions of each project, but we only studied a single revision of each project, and hence, it is possible that DeFlaker had not identified any flaky tests in that revision. The iDFlakies dataset consists of both order dependent tests (which are detected by shuffling execution orders), and non-order dependent tests (which are flaky regardless of execution order). Since we purposefully did not shuffle the execution order of the tests (as described above), we include only the non-order dependent tests from iDFlakies for comparison.

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

**Study Summary:** The results from this study demonstrate that flaky tests are extremely difficult to detect. If developers would like to try to find all tests that are flaky in their test suite, they likely should consider re-running those tests thousands of times: only half of the flaky tests that we found failed more than 100 times out of the 10,000 runs. Moreover, we know that even running tests 10,000 times will *still* not guarantee that all flaky tests have been found, since we did not succeed in reproducing many flaky test failures observed in prior work. This study underscores the need for approaches that detect flaky tests without rerunning them.

## III. Approach

Our primary goal with FlakeFlagger is to create a new approach to proactively identify which tests in a test suite are flaky, before they become a nuisance and *without* rerunning them many times. For instance, when a new test is introduced, developers might use FlakeFlagger to identify if that new test is likely to become flaky — since research has shown that most flaky tests are flaky when they are introduced [27]. Alternatively, developers might use FlakeFlagger to aid in a search for flaky tests in a large test suite, where developers identify that a portion of the test suite is or is not flaky, and use FlakeFlagger to help label the rest of the tests as flaky or not. Researchers constructing new approaches to detect and repair flaky tests need large datasets of flaky tests, and FlakeFlagger is perfectly suited to help find these flaky tests faster. Instead of re-running *every* test in our dataset 10,000 times, if we already had trained FlakeFlagger on a subset of the tests or projects, we could have greatly reduced the execution time needed to rerun these tests by only re-running those reported by FlakeFlagger as likely to be flaky. Note that FlakeFlagger is orthogonal to techniques that mutate the execution environment in order to increase the likelihood of a test appearing flaky: these tools are still expensive to run, and FlakeFlagger can help focus computing time on the tests most likely to be flaky. Even if FlakeFlagger presents a false positive (i.e., a test FlakeFlagger declares to be flaky, even though it is not), our approach can help developers reduce

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

| | Feature | Description |
|---|---|---|
| Test Smells | Indirect Testing | True if the test interacts with the object under test via an intermediary [24] |
| | Eager Testing | True if the test exercises more than one method of the tested object [24] |
| | Test Run War | True if the test allocates a file or resource which might be used by other tests [24] |
| | Conditional Logic | True if the test has a conditional if-statement within the test method body [25] |
| | Fire and Forget | True if the test launches background threads or tasks. [26] |
| | Mystery Guest | True if the test accesses external resources [24] |
| | Assertion Roulette | True if the test has multiple assertions [24] |
| | Resources Optimism | True if the test accesses external resources without checking their availability [24] |
| Numeric Features | Test Lines of Code | Number of lines of code in the test method body |
| | Number of Assertions | Number of assertions checked by the test |
| | Execution Time | Running time for the test execution |
| | Source Covered Lines | Number of lines covered by each test, counting only production code |
| | Covered Lines | Total number of lines of code covered by the test |
| | Source Covered Classes | Total number of production classes covered by each test |
| | External Libraries | Number of external libraries used by the test |
| | Covered Lines Churn | $h$-index capturing churn of covered lines in past 5, 10, 25, 50, 75, 100, 500, and 10,000 commits. Each value $h$ indicates that at least $h$ lines were modified at least $h$ times in that period. |

the candidate number of tests they need to investigate via re-running or manual inspection.

By proactively identifying flaky tests, we may also help developers understand why these tests are flaky. Prior work has suggested different properties of tests that might make them more likely to be flaky, and FlakeFlagger can report which of these features are present in each test [4], [8]. In practice, if a feature has a strong correlation with flakiness, developers might choose to focus on this feature in their future test maintenance and development activities.

Figure 1 shows a high-level overview of our approach to detect flaky tests. First, we collect a series of features describing each test in a project. In a developer's scenario, we would assume that some of these tests would be known to be flaky, and the developers would be interested in detecting other flaky tests. For instance, the developer might know the flaky tests that exist in their test suite currently, and would like to identify if a newly written test is flaky. We collect behavioral features (such as API usage) of each test, and then construct a classifier to predict which tests are flaky. In a controlled experiment (with known flaky tests), we can perform cross-validation, and generate a confusion matrix that describes the performance of the classifier. In practice, without an oracle, we would present a report to developers including a list of likely flaky tests.

### A. Prediction Features

To develop a list of features that may be predictive of flakiness, we look to prior flaky test research [1], [4], [6], [8]. Ahmed et al. [8] categorized 23 developer-reported factors which affect test flakiness. These features are described by practitioners at a high level, and include test case complexity, hard-coded values and test smells. Eck et al. [4] interviewed 21 developers about flaky tests and tabulated the frequency of different kinds of flaky tests as well as developers' fixes for those flaky tests. Whereas prior flakiness classification approaches used static, code-level features (e.g. presence of textual tokens in the body of each test method), these surveys describe features that are more nuanced. For instance, Eck et al. note that many flaky tests are flaky due to causes not in the test method itself, but instead, in the production code that is executed by that test [4].

Inspired by previous studies on test flakiness, we developed a list of sixteen features, some of are based on general studies on the causes of flaky tests [1], [8], while others are defined as bad practices in writing unit tests [28]. Unfortunately, some of these flaky test root cases are too complicated to detect without human intervention, for instance, Eck et al's "Too restrictive range" (which effectively describes the case where an assertion is wrong). Hence, we considered all of the features described in the prior works, and then selected only those for which we could write automated detectors. We implemented detectors for each of the features shown in Table II.

While some of the features can be detected by inspecting the test method statically (specifically, the conditional logic smell and test line of code), the rest of the features require more than static analysis. For instance: existing automated test smell detectors have analyzed *only* the code that is present in a test method [29]–[32], and hence can fail to label tests that are smelly because they invoke a helper method, which in turn, performs smelly behavior. For instance, the "Fire and Forget" smell exists in tests that spawn background threads or tasks; a smell detector that considered only direct calls to launch threads in the test method body would not define a test as smelly if it called a helper method to launch that background thread. Since our goal is not to precisely detect test smells (as identified by humans), but rather, to find features that may be representative of flaky tests, we decided to expand our definition of many of these smells to be inclusive of all code executed by a test, rather than just the code contained in the test method body itself.

We developed a hybrid static/dynamic framework to collect the statement coverage of each test, and then statically analyze the covered code in order to collect these behavioral features. For instance, we determined that a test had the "fire and forget" smell feature if, anytime during its execution (including in the production code that is called by the test), that test launched a background thread or task. We also collect a variety

of other features related to the statement coverage of each test, such as how many recently changed lines of code are covered. We implemented this framework as an extension to the Maven build system, allowing for a zero-configuration feature collection process that automatically parses the target project's build scripts to modify them to collect coverage and report features. Due to space limitations, we omit additional details on precisely how each feature is detected, and make our entire implementation publicly available [20], [21].

While we have built FlakeFlagger to analyze Java code using the Maven build system, our approach is sufficiently general to be applied to other build systems or languages. This list of features is not intended to be complete: there may yet be other features that can be easily collected and will be useful for predicting test flakiness. However, we empirically found that this set of features yielded good prediction performance for FlakeFlagger (Section IV). Again, by making our dataset and infrastructure publicly available, we enable future research on behavioral features of flaky tests.

### B. Flaky Tests Classifier

For our classifier, each test instance *i* is represented as a vector $\{x_1, x_2, x_3, \ldots, x_n\}$ where each *x* represents a feature value and *n* correspond the total number of features. To avoid inaccuracies in training, we perform an automated data inspection and cleaning process as follows:

Instances with missing values (e.g., missing features) can have a negative impact on model performance [33]. Since we use different approaches to collect different features, it is possible for some features to be missing for some tests. For instance: if a test crashes in the middle of its execution (e.g., with an unrecoverable out-of-memory exception), we will be unable to collect telemetry from that test, and hence, unable to calculate several of our features. Some tests are not written in Java, and hence the feature detectors may not be applicable to them, and due to inheritance, some tests may not have source code in the project under test (with the test residing in a 3rd party library). We found this occurred fairly infrequently (roughly 1% of the tests - the difference between Tables I and III) and handle tests with at least one missing value by excluding them from our experiment (future work could support such tests).

Representing the range of numeric values of each feature is also very important. Some features have discrete values — for instance, each smell-related feature has a boolean value (smelly or not). However, there are also features with continuous values shown in Table II, which do not have a maximum value, and may need to be discretized [34]. From a model learning perspective, discretization is crucial because discretized features have a better chance of correlating with the class value as the range of values are limited, which helps features to be interpreted [35]. We considered two main approaches to discretization. First, we considered values as they are, without applying any discretization techniques. We consider this option because we expect to not have long ranges of continuous data for most features. The second approach we considered is to divide continuous data into a fixed number of intervals, called bins, which is a common techniques to discretize continuous data [34].

Our approach is agnostic to the classification algorithm that we use to build the model. Hence, rather than build a single model, we designed FlakeFlagger to construct a set of models based on seven different supervised learning algorithms: Decisions Tree (DT), Random Forest (RF), Support Vector Machine (SVM), Multilayer Perceptron (MLP), Naive Bayes (NB), Adaboosting (Ada), and K-Nearest Neighbor (KNN), using the Scikit-learn package [36]. This selection of models builds on flaky test classification prior work that considered DT, RF, SVM, KNN and NB models only, which found that RF performed best [12]. In our evaluation (described in the following section), we also found that the Random Forest had the best performance, and report results only for this model, but make all models available in our artifact [20], [21]. We perform a feature selection process using *information gain*, which computes the amount of information that a feature can provide for a classification [37]. The range of information gain is between 0 and 1, where higher values indicate more predictive power. Imbalanced datasets (where there is not an equal number of instances in each class — flaky and non-flaky tests in our case) usually have very low information gain values. We compute the information gain for each feature, and include only features with an information gain of at least $0.01$, following previous work [12].

## IV. Evaluation

To evaluate our classifier, we designed experiments to answer the following research questions:

**RQ1:** How effective is FlakeFlagger at predicting flaky tests?

**RQ2:** How helpful is each feature in distinguishing between flaky and non flaky tests?

### A. Experimental Design

To evaluate FlakeFlagger, we used the extensive dataset described in Section II as a source of flaky tests. To collect the various features needed to predict flakiness, we re-executed each test one more time (using the feature extractor) using the same environment.

Machine learning classifiers rely on two data sets: one to build the model (training) and another for testing. We apply $k$-fold cross validation [38], [39] to evaluate our model. However, $k$-fold cross validation is most applicable to data that is evenly balanced, where the proportions of each class (in our case: flaky and not flaky) are similar. Balanced datasets reduce the risk of any of the $k$ folds having only a single (or no) instances of one of the classes (flaky and not flaky).

Since most tests are not flaky, we have imbalanced data, and hence, have designed FlakeFlagger to use two data sampling techniques: *SMOTE* [40] and *random undersampling* [41], which we also compare to a baseline without sampling. We perform this sampling *only* when training the model, and *not* when testing it, to ensure a valid and fair result. We train and test our models considering all of the tests from all of the projects in a single bucket, randomly shuffling which tests

from which project appear in each fold. To evaluate the effect that the size of the testing and training dataset has on the classification results, we consider both 10% and 20% testing data sizes. Following past work [12], we consider only features with an information gain of at least 0.01, and report the information gain of each feature.

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

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

### B. RQ1: Efficacy of our Classifier

Following the evaluation procedure described in the previous section, we trained and tested FlakeFlagger. We created several models, consider 2 approaches to discretization, 2 sizes of training sets, 3 data balancing approaches, and 7 different classification algorithms. Due to space limitations, we show only the results of the single best model configuration: a random forest model built using the SMOTE technique for balancing the training data (and using unbalanced testing data), no discretization and a 90-10 training-testing split (our artifact includes all results [21]). We found that SMOTE worked better than other sampling techniques due to the relatively small ratio of flaky tests to the total number of tests. Random forest was far more effective than the other algorithms, but we did not find a meaningful difference in performance between a 10% and 20% testing data size.

In order to evaluate the performance of our flaky test classifier, we compared its prediction results to the state-of-the-art flaky test classifier, a vocabulary-based approach proposed by Pinto et al. [12]. This approach extract tokens from each test using a simple bag-of-words model, under the theory that flaky tests may use similar APIs, keywords, etc., and hence, tests can be classified as flaky or not based on the presence of various tokens (the model also uses one non-token feature: the length of the test method). We also considered a hybrid model that adds the token features to FlakeFlagger's features. Note that our evaluation methodology differs somewhat from the prior approach, in which the authors trained *and* tested their model on a balanced flaky/non-flaky dataset with the exact same number of flaky and non-flaky tests: we consider a more realistic scenario where the model is trained on a balanced flaky/non-flaky dataset, but tested on the complete set of tests (which is not balanced) [12]. From a methodological perspective, balancing the number of flaky and non-flaky tests by ignoring tests from the majority (non-flaky) class may result in over approximating the performance of the model, if this balancing is done during the testing phase. This is a particularly important distinction on projects with very large numbers of tests such as *incubator-dubbo*, with 2,174 tests of which only 19 were flaky: our cross-validation approach would test the model on a all of the 2,174 tests, whereas the methodology used in [12] would test the model on a much smaller set of non-flaky tests. In our evaluation, we use stratified cross-validation.

Table III presents the results of this experiment, showing the true positives, false negatives, false positives, true negatives, precision, recall and F1 score for each approach broken down by project. Even though the models are not trained and tested by project (i.e. the project name is not a feature in the model), after running the evaluation we mapped each test back to its project to allow us to inspect how the models performed on each project. We show the aggregate precision, recall, F1 score, and AUC. FlakeFlagger is a learning-based approach, and hence, we expect that it might perform better on projects that have many flaky tests (providing more training data that could help the classifier to better predict flaky tests in that project) than those with very few. To avoid biasing our dataset towards projects with the most flaky tests, we did not impose a threshold for a minimum number of flaky tests to include the project in our evaluation, including all projects with at lest one flaky test. We do not include the 298 tests for which we were unable to collect all features (typically due to the tests having non-Java components).

Overall, FlakeFlagger and the vocabulary-based approach both detected a very similar number of flaky tests (599 and 583 respectively, out of a total of 808 flaky tests), but the two approaches varied dramatically in terms of precision — FlakeFlagger had a far lower false positive rate with just 406, compared to 4,683 false positives from the vocabulary-based approach. We found that combining FlakeFlagger's features with the tokens from the vocabulary-based approach yielded slightly improved performance. The following section explores which features from the vocabulary-based approach contributed to this improvement.

Considering our initial use-case of a researcher or developer using FlakeFlagger to determine which tests to run time-intensive flaky test detectors on, using either FlakeFlagger or the vocabulary-based approach would result in roughly the same number of flaky tests eventually detected (that is, both have comparable recall). However, the total amount of time

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

| | | Flaky by | FlakeFlagger | | | | | | | Vocabulary-Based Approach [12] | | | | | | | Combined Approach | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Project** | **Tests** | **Reruns** | **TP** | **FN** | **FP** | **TN** | **Pr** | **R** | **F** | **TP** | **FN** | **FP** | **TN** | **Pr** | **R** | **F** | **TP** | **FN** | **FP** | **TN** | **Pr** | **R** | **F** |
| spring-boot | 2,108 | 160 | 139 | 21 | 15 | 1,933 | 90% | 87% | 89% | 134 | 26 | 703 | 1,245 | 16% | 84% | 27% | 143 | 17 | 18 | 1,930 | 89% | 89% | 89% |
| hbase | 431 | 145 | 129 | 16 | 32 | 254 | 80% | 89% | 84% | 89 | 56 | 152 | 134 | 37% | 61% | 46% | 130 | 15 | 33 | 253 | 80% | 90% | 84% |
| alluxio | 187 | 116 | 116 | 0 | 0 | 71 | 100% | 100% | 100% | 108 | 8 | 11 | 60 | 91% | 93% | 92% | 116 | 0 | 0 | 71 | 100% | 100% | 100% |
| okhttp | 810 | 100 | 52 | 48 | 159 | 551 | 25% | 52% | 33% | 79 | 21 | 444 | 266 | 15% | 79% | 25% | 46 | 54 | 104 | 606 | 31% | 46% | 37% |
| ambari | 324 | 52 | 47 | 5 | 3 | 269 | 94% | 90% | 92% | 36 | 16 | 121 | 151 | 23% | 69% | 34% | 47 | 5 | 3 | 269 | 94% | 90% | 92% |
| hector | 142 | 33 | 30 | 3 | 8 | 101 | 79% | 91% | 85% | 13 | 20 | 23 | 86 | 36% | 39% | 38% | 25 | 8 | 11 | 98 | 69% | 76% | 72% |
| activiti | 2,043 | 32 | 10 | 22 | 43 | 1,968 | 19% | 31% | 24% | 12 | 20 | 531 | 1,480 | 2% | 38% | 4% | 7 | 25 | 34 | 1,977 | 17% | 22% | 19% |
| java-websocket | 145 | 23 | 19 | 4 | 1 | 121 | 95% | 83% | 88% | 23 | 0 | 74 | 48 | 24% | 100% | 38% | 19 | 4 | 4 | 118 | 83% | 83% | 83% |
| wildfly | 1,023 | 23 | 11 | 12 | 27 | 973 | 29% | 48% | 36% | 20 | 3 | 554 | 446 | 3% | 87% | 7% | 17 | 6 | 24 | 976 | 41% | 74% | 53% |
| httpcore | 712 | 22 | 14 | 8 | 23 | 667 | 38% | 64% | 47% | 16 | 6 | 375 | 315 | 4% | 73% | 8% | 15 | 7 | 24 | 666 | 38% | 68% | 49% |
| logback | 805 | 22 | 3 | 19 | 17 | 766 | 15% | 14% | 14% | 10 | 12 | 259 | 524 | 4% | 45% | 7% | 5 | 17 | 11 | 772 | 31% | 23% | 26% |
| incubator-dubbo | 2,174 | 19 | 8 | 11 | 35 | 2,120 | 19% | 42% | 26% | 11 | 8 | 813 | 1,342 | 1% | 58% | 3% | 13 | 6 | 23 | 2,132 | 36% | 68% | 47% |
| http-request | 163 | 18 | 12 | 6 | 6 | 139 | 67% | 67% | 67% | 16 | 2 | 84 | 61 | 16% | 89% | 27% | 12 | 6 | 6 | 139 | 67% | 67% | 67% |
| wro4j | 1,135 | 16 | 4 | 12 | 2 | 1,117 | 67% | 25% | 36% | 2 | 14 | 101 | 1,018 | 2% | 12% | 3% | 0 | 16 | 1 | 1,118 | 0% | 0% | 0% |
| orbit | 86 | 7 | 1 | 6 | 8 | 71 | 11% | 14% | 12% | 6 | 1 | 32 | 47 | 16% | 86% | 27% | 1 | 6 | 7 | 72 | 12% | 14% | 13% |
| undertow | 183 | 7 | 2 | 5 | 8 | 168 | 20% | 29% | 24% | 6 | 1 | 63 | 113 | 9% | 86% | 16% | 3 | 4 | 8 | 168 | 27% | 43% | 33% |
| achilles | 1,317 | 4 | 2 | 2 | 3 | 1,310 | 40% | 50% | 44% | 0 | 4 | 0 | 1,313 | 0% | 0% | 0% | 0 | 4 | 0 | 1,313 | 0% | 0% | 0% |
| elastic-job-lite | 558 | 3 | 0 | 3 | 0 | 555 | 0% | 0% | 0% | 0 | 3 | 34 | 521 | 0% | 0% | 0% | 1 | 2 | 0 | 555 | 100% | 33% | 50% |
| zxing | 345 | 2 | 0 | 2 | 2 | 341 | 0% | 0% | 0% | 1 | 1 | 144 | 199 | 1% | 50% | 1% | 0 | 2 | 2 | 341 | 0% | 0% | 0% |
| assertj-core | 6,261 | 1 | 0 | 1 | 5 | 6,255 | 0% | 0% | 0% | 0 | 1 | 6 | 6,254 | 0% | 0% | 0% | 0 | 1 | 0 | 6,260 | 0% | 0% | 0% |
| commons-exec | 55 | 1 | 0 | 1 | 1 | 53 | 0% | 0% | 0% | 1 | 0 | 18 | 36 | 5% | 100% | 10% | 0 | 1 | 1 | 53 | 0% | 0% | 0% |
| handlebars.java | 420 | 1 | 0 | 1 | 5 | 414 | 0% | 0% | 0% | 0 | 1 | 91 | 328 | 0% | 0% | 0% | 0 | 1 | 0 | 419 | 0% | 0% | 0% |
| ninja | 307 | 1 | 0 | 1 | 3 | 303 | 0% | 0% | 0% | 0 | 1 | 50 | 256 | 0% | 0% | 0% | 0 | 1 | 0 | 306 | 0% | 0% | 0% |
| **Total** | 21,734 | 808 | 599 | 209 | 406 | 20,520 | 60% | 74% | 66% | 583 | 225 | 4,683 | 16,243 | 11% | 72% | 19% | 600 | 208 | 314 | 20,612 | 66% | 74% | 86% |
| **AUC** (Average per fold) | | | 86% | | | | | | | 75% | | | | | | | 86% | | | | | | |

needed to run such an experiment would vary dramatically between the two models, since FlakeFlagger had far fewer false positives (406 vs 4,683). Assuming that each test would take a comparable amount of time to run flaky test detectors on, our developer (or researcher) would be able to confirm the flakiness of FlakeFlagger's 1,005 reported flaky tests (599 TPs plus 406 FPs) in roughly 18% of the time that it would take to confirm the flakiness of the 5,266 reported flaky by the vocabulary-based approach (583 TPs plus 4,683 FPs).

We were initially surprised that the precision of the vocabulary-based approach's was so much lower than FlakeFlagger's, and indeed, lower than reported by the original authors [12]. We found that the tokens used in Pinto et al.'s bag-of-words model did indeed frequently occur in flaky tests, but also occurred quite frequently in non-flaky tests. For example, one of the most relevant tokens that the model relied upon (both in our study, and in [12]) was the Java `throws` keyword. However, when examining the entire corpus, we discovered that this keyword is used quite frequently in both flaky and non-flaky tests, and hence, is not a very good predictor of flakiness.

FlakeFlagger's performance varied across projects: on some projects (e.g., alluxio), we had perfect precision and recall, while on others (e.g., okhttp and activiti) the approach was less successful. We investigated more closely the different factors that could cause such a varied performance among different projects. The first and most obvious factor is the size of the training data: our model performed best on the two projects which had the most known flaky tests (alluxio and spring-boot each had more than 100). On projects with very few known flaky tests (less than 4), FlakeFlagger did not classify *any* of the flaky tests as flaky, resulting in F1 scores of 0. This results from the lack of training data that are representative of the flaky tests in these projects. However, note that even on these projects with so few flaky tests (e.g., zxing with only two known flaky tests, ninja with only one), even though FlakeFlagger failed to identify the flaky tests (true positives), it had far fewer false positives than the other approach.

More broadly speaking, we can attribute the variation of prediction performance between projects to the relative generality of our features (such as test execution time, coverage of recently changed lines, etc.). Each project has its own environmental assumptions, development patterns, and other unique characteristics that can make it difficult to create a single general-purpose approach to classifying tests as flaky or not. Another explanation for why performance varies across projects may be that not all flaky tests have been labeled correctly — no rerun-based technique can guarantee to find all flaky tests (even after 10,000 reruns). That is: there may be tests that are labeled as "not flaky" in our dataset that are in fact flaky, but we simply did not observe any flaky failure of those tests in our experiments.

The higher number of observed flaky tests in a single project does not guarantee that FlakeFlagger performs well. Some flaky failures are due to rare dependency conflicts and network failures that are not captured well from our features described

TABLE IV: List of top 23 features by information gain (IG) for FlakeFlagger and the vocabulary-based approach. Our models only include features with IG $\geq 0.01$.

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

in Table II. For example, we notice that okhttp has a high number of false positives and false negatives. With a further inspection on this particular project, we found that a group of tests had all failed in the same way due to the same dependency problem in one single run. However, only *some* tests that used this dependency presented a flaky failure to us, and hence, we observed many false positives from our model, with other (not flaky) tests that use this same dependency classified to be flaky. In this same project, our model also showed a large number of false negatives (tests that are flaky but predicted as not flaky). Upon further investigation, we found that these tests were flaky due to transient network-related failures that occurred when they were run. While our model *does* consider usage of network APIs (via the "mystery guest" smell), we found that *most* tests that used network APIs were not labeled as flaky in our dataset because we did not witness a network failure in the 10,000 runs of those tests. As a result, the few tests that *did* fail due to network failures were typically labeled as *not flaky* by the model, resulting in false negatives. We discuss these limitations further in Section V.

### C. RQ2: How useful is each feature?

To gain more insight into which textual and behavioral features are correlated with test flakiness, we report the information gain of each feature in FlakeFlagger's model, and the top 23 features in the model built using the vocabulary-based approach.

Table IV lists all of FlakeFlagger's features sorted by information gain, along with the top 23 features used by the comparison system based on the vocabulary-based approach [12]. Note that we used only features with an information gain of at least $0.01$ in our models, but we include all top 23 features in this table. Overall, we found that features that considered dynamic behavior from each test (e.g., execution time, covered lines, and coverage of recently changed lines) had a far greater information gain than the tokens that were statically extracted from the test method bodies. None of the test smells that we collected had a strong information gain, which may indicate that test smells are not well-correlated with test flakiness — all of the smells had an information gain less than our threshold of $0.01$, and in fact, "eager testing" had an information gain of less than $0.001$. We found that the top eight FlakeFlagger features each had a higher information gain than the highest gain vocabulary feature.

The most effective features (execution time, various forms of coverage, size of test and usage of external APIs) are all measures of test case complexity. This confirms industry reports that correlate test size with test flakiness [42]. For instance: it is perhaps likely that small unit tests are less likely to be flaky than large integration tests — and this behavior is captured by both the execution time of each test, and also by the coverage of each test. Due to varying development styles, other behavioral features likely have high variance between projects (without signaling flakiness). For instance, "Number of Assertions" likely varies more between projects because of coding conventions more than it varies between flaky and non-flaky tests. Future work might consider ways to normalize these features by-project to account for such variance, and we make all of our tools and data available to allow other researchers to join in this effort.

In the model built using the vocabulary-based approach [12], the features with the highest information gain were: test lines of code, presence of the 'throws' Java keyword, and several tokens like 'should', 'exception', and 'mtfs', each with an information gain significantly lower than the top features in FlakeFlagger's model. We believe that these tokens might be over-fitting to specific patterns in some of the flaky tests in our dataset, and indeed, these top three tokens differ from those reported by Pinto et al. in their initial work [12]. Table V directly compares the frequency of those three tokens in flaky and other (non-flaky) tests reported by the vocabulary-based approach [12] and also their frequencies in the dataset that we used in this study. While the prior study found that these tokens occurred far more frequently in flaky tests than in non-flaky tests, when looking at our corpus, we found that these tokens occurred far more frequently in non-flaky tests than in flaky tests.

The majority of the flaky tests in the prior study with the 'job' token came from a single project, "oozie," which we did not study in our evaluation. At the same time, the majority of non-flaky tests with the token 'job' in our dataset were in the project "elastic-job-lite," which was not included in the prior evaluation. Clearly, the co-occurrence of individual tokens with flaky tests can vary dramatically between projects. Terms that correlate with flakiness in one project can not be expected to correlate with flakiness in other projects — this is also evident from the limited number of projects which contain each token.

Note that this finding only underscores the need for a large, balanced dataset of flaky tests (like the one we collected and

TABLE V: Comparison showing the predictive power of the top 3 tokens reported by the vocabulary-based approach evaluation on the DeFlaker dataset [12] compared to this evaluation on our dataset. For each token, we show the information gain (IG), the number of flaky tests containing that token (number of projects containing those tests in parentheses), and the number of other, non flaky tests containing that token (with the number of projects containing those tests).

| | Vocabulary-Based Evaluation [12] | | | | | This Evaluation | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| **Token** | **IG** | **Flaky** | | **Not Flaky** | | **IG** | **Flaky** | | **Not Flaky** | |
| job | 0.2053 | 524 | (2) | 4 | (1) | 0.00044 | 48 | (4) | 703 | (8) |
| table | 0.1449 | 406 | (4) | 8 | (2) | 0.00819 | 114 | (4) | 405 | (11) |
| id | 0.1419 | 522 | (9) | 52 | (4) | 0.00037 | 125 | (9) | 2,429 | (19) |

described in Section II): the DeFlaker dataset that Pinto et al. used contained *more* flaky tests than our dataset (1,403 vs 810). However, a single project in that dataset ("oozie") contributed more than half of those flaky tests (856), which can make it extremely difficult to draw conclusions that can generalize beyond a single project, or beyond the dataset. We look forward to contribute to new community efforts to build shared flaky test datasets [43], making all of the log files from each of the 10,000 executions of each test suite in our study publicly available in our artifact [21].

## V. Discussion and Threats to Validity

While we are confident in our findings, it is nonetheless worthwhile to acknowledge various threats to the validity of our conclusions. For instance, the list of projects in our prediction experiment may not be representative of all projects, limiting the generalizability of our conclusions. We reduce this threat by selecting different projects from different domains. Further, our project list is based on those studied by others.

RQ1 showed that our classifier was able to predict 599 out of 808 flaky tests. However, note that our ground truth is imperfect: some of the false positives may in fact be true positives, where the failure is in the rerun process to identify that the test is flaky. Unfortunately, it may never be possible to provably identify all flaky tests in a project. As we witnessed in our empirical study (Section II), the environment in which a test runs may affect if the test appears flaky. Since on the same revisions of the same projects, we found a different set of flaky tests than were found by the iDFlakies and DeFlaker authors —- even after re-running those tests far more times than the original authors — we recognize this as a limitation.

It may be possible to find more flaky tests by injecting artificial non-determinism into the execution of each test, for instance, arbitrarily slowing the execution of each test or shuffling test orders. These other approaches may find more flaky tests of one particular type: for instance, reordering tests will expose more order dependent flaky tests, perturbing timing may expose more concurrency-related flaky tests. However, a significant concern with any such approach is that it will result in a dataset that is overly representative of those specific sources of flakiness. For example, Luo et al.'s study found that just 13% of flaky tests were caused by order dependencies [1]: an approach that built a dataset of primarily tests flaky due to this cause would likely not represent all flaky tests.

Similarly, we observed that some (but far from most) tests that made calls through the network could fail due to flakiness in the event of a network failure. Hence, one approach to find *more* flaky tests could be to inject network failures while tests run. However, we speculate that the result of this experiment would likely be that *all* tests that make network calls will be labeled as flaky. Such an approach would likely not be very useful for developers, who presumably are already aware that tests that rely on network resources might fail if the network is unreliable. This mirrors the sentiment expressed by Harman and O'Hearn suggesting that based on Facebook's experiences, we should "assume that all tests are flaky" [14] and Google, where all tests that execute in more than a single thread are assumed to be potentially flaky [44]. Hence, we remain confident that our 10,000-rerun based flaky test detection approach represents the best-possible approach to build our dataset: one that includes quite literally, the flaky tests that developers might have observed in the course of normal development, without injecting additional non-determinism.

We considered multiple supervised learning algorithms and found that Random Forest resulted in the best performance on our dataset. This result may not hold on other datasets. Similarly, we found that some features (like execution time, and coverage of recently changed lines) are more predictive of flakiness than others (like test smells), and these findings may not hold on other datasets. We include all results and scripts in our artifact, and encourage other researchers to experiment with different configurations of FlakeFlagger [21].

Our evaluation of FlakeFlagger presents a direct comparison to a state-of-the-art flaky test prediction approach [12], but does not include a direct comparison with other flaky test detection tools like DeFlaker [6], iDFlakies [7] and NonDex [16]. These other approaches rely on repeatedly re-running a test (perhaps modifying the environment that the test runs in) in order to detect if that test is flaky. Section II describes our empirical study to identify the prevalence of infrequently-failing tests, and found that hundreds of flaky tests could only be detected after an enormous number of reruns, or using a prediction model like FlakeFlagger. Future work might combine these two approaches, using FlakeFlagger to prioritize which tests to apply these rerun-based flaky test detectors to.

To increase our confidence in our implementation, we implemented our classifier using the supervised learning algorithms provided by *scikit-learn*. This well-regarded library provides the implementations of different techniques to balance data, impute missing values and compute feature correlations. Nonetheless, there may be bugs in the tooling or how we used it; we make best efforts to check our results and will provide a replication package. We implemented our own test smell detectors rather than using existing detectors. This was largely due to the lack of a published, experimentally evaluated test smell detector for Java at time of writing. Future work might consider extending our test smells detector, perhaps integrating

the recently released *tsDetect* smell detector [45].

Even with a replication package, some of our our results may be difficult to reproduce, since we are purposefully experimenting on non-deterministic systems. In particular, when rerunning tests (in order to find flaky tests), there is no guarantee that a flaky test will fail. To aid future researchers, we collected the build logs of each test execution and include these in our artifact [21].

## VI. Related Work

Several recent works have focused on the problem of detecting and managing flaky tests. Luo et al. provided an empirical study of flaky tests by investigating commits logs of open-source projects [1]. They presented the most common causes of flaky tests and described possible strategies to fix flaky tests. Unlike our dataset of flaky tests (which is constructed by rerunning tests), Luo et al.'s dataset is constructed by analyzing developer reports. DeFlaker is an approach to detect flaky tests immediately after the test fails (without rerunning it) by monitoring the coverage of the latest code changes [6]. In contrast, FlakeFlagger's detects flaky tests before they fail.

More similar to our approach, iDFlakies aims to proactively detect flaky tests (before they cause unexpected failures) by rerunning tests in different orders [7]. This approach is particularly suited to detect order-dependent flaky tests (those which can fail deterministically modulo their execution order) [23], and reruns tests 100 times. In comparison, FlakeFlagger aims to find flaky tests regardless of their underlying source of non-determinism, and in our empirical study to detect flaky tests, we reran tests two orders of magnitude more (10,000 times total) than the iDFlakies work (100 times). Lam et al. later re-ran tests from this dataset 4,000 times in order to evaluate how difficult these failures were to reproduce, finding a mean failure rate of 2.7% [22]. Order-dependent flaky tests have received quite a bit of attention in general, and several techniques have been proposed to detect them [23], [46]–[48], to isolate tests to avoid order dependencies [3], and to repair tests to remove those dependencies [49]. Similarly, other specialized kinds of flaky tests, like those that are flaky due to commonly incorrect assumptions of API behavior can be detected through specialized means [16]. Lam et al. studied 315 real flaky test failures at Microsoft, and found that for the majority of them, the failure that occurred on the build server could not be reproduced on a developer machine, even after trying to rerun the test 100 times [17]. This finding is similar to ours: we only reproduced half of the flaky test failures that prior work found on the same projects, even though we reran the tests 10,000 times. Strandberg et al. studied intermittently failing tests for embedded systems, finding that in this domain, flakiness was typically caused by environmental factors [50].

Eck et al. analyzed flaky tests by conducting a survey of developers to understand the root cause of flakiness [4]. This survey helped inspire our feature selection. Harman and O'Hearn [14] emphasize the importance of flakiness on software testing, suggesting that we assume that *all* test are flaky, and calling for tools and techniques to automatically assess the likelihood of a new test becoming flaky in the future. FlakeFlagger answers this call directly, by helping developers identify which tests are likely to be flaky based on the behavior of other tests. Several recent works have begun study the impact of flakiness on other software testing practices, like mutation testing [51] and program repair [52].

Most similar to our approach are prior works that also aim to predict flakiness [5], [9], [15], with the most relevant being Bertolino et al.'s *FLAST* [13], [19] and Pinto et al.'s flaky test vocabulary work [12]. Unlike FlakeFlagger, both FLAST and Pinto et al.'s approach considers only the text in the test method body itself, and does not consider any features of the code that is called by that test, or code that might exist in setup or teardown methods. In our evaluation, we found that FlakeFlagger significantly outperformed vocabulary-based approaches in predicting flaky tests.

More generally, machine learning techniques have been adopted widely in software testing [53]–[59]. Durelli et al. [58] showed that ML algorithms are commonly adopted to automate software testing tasks. Azeem et al. [54] proposed a systematic literature review on the ML techniques used in code smell detection. Nucci et al. [57] discussed possible limitations which may result in unexpected result in code smell detection by mainly analyzing Fontana et al.'s [55] work detecting code smells using ML. We did not use ML to detect test smells, but used traditional source code parsing to detect smells [29]–[32]. It would be interesting to compare the performance of our hybrid static/dynamic test smell detectors with prior techniques, using a human oracle to determine valid smells.

## VII. Conclusion

Flaky tests make testing unreliable because it is hard to say which test failures represent true regressions. We presented a very large empirical study showing just how difficult it is to proactively detect flaky tests by rerunning them. FlakeFlagger is a new approach for automatically classifying tests as flaky or not *without reruns*, and improves on existing classifiers by collecting behavioral features from each test. Developers can use FlakeFlagger to determine if a newly added test is likely to be flaky, or could use it to help classify existing tests. Developers and researchers alike can use FlakeFlagger to focus more precise (and costly) flaky test detection efforts on the tests most likely to be flaky. Compared to a state-of-the-art flaky test classifier, FlakeFlagger had 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.

## VIII. Data Availability

All tools and data produced for this paper are publicly and permanently archived at [21].


## Acknowledgements

Jonathan Bell's group is supported in part by NSF CCF-2100037 and CNS-2100015. We thank Paul Ammann, Wing Lam and Darko Marinov for discussions about this work.

## REFERENCES


[1] Q. Luo, F. Hariri, L. Eloussi, and D. Marinov, "An empirical analysis of flaky tests," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2014, pp. 643–653.

[2] "Tott: Avoiding flakey tests," Apr 2008. [Online]. Available: https://testing.googleblog.com/2008/04/tott-avoiding-flakey-tests.html

[3] J. Bell and G. Kaiser, "Unit test virtualization with vmvm," in *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE 2014. New York, NY, USA: Association for Computing Machinery, 2014, pp. 550–561. [Online]. Available: https://doi.org/10.1145/2568225.2568248

[4] M. Eck, F. Palomba, M. Castelluccio, and A. Bacchelli, "Understanding flaky tests: The developer's perspective," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2019. New York, NY, USA: Association for Computing Machinery, 2019, pp. 830–840. [Online]. Available: https://doi.org/10.1145/3338906.3338945

[5] K. Herzig and N. Nagappan, "Empirically detecting false test alarms using association rules," in *Proceedings of the 37th International Conference on Software Engineering-Volume 2*. IEEE Press, 2015, pp. 39–48.

[6] J. Bell, O. Legunsen, M. Hilton, L. Eloussi, T. Yung, and D. Marinov, "Deflaker: Automatically detecting flaky tests," in *Proceedings of the 40th International Conference on Software Engineering*, ser. ICSE '18. New York, NY, USA: Association for Computing Machinery, 2018, pp. 433–444. [Online]. Available: https://doi.org/10.1145/3180155.3180164

[7] W. Lam, R. Oei, A. Shi, D. Marinov, and T. Xie, "idflakies: A framework for detecting and partially classifying flaky tests," in *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*. IEEE, 2019, pp. 312–322.

[8] A. Ahmad, O. Leifler, and K. Sandahl, "Empirical analysis of factors and their effect on test flakiness-practitioners' perceptions," *arXiv preprint arXiv:1906.00673*, 2019.

[9] T. M. King, D. Santiago, J. Phillips, and P. J. Clarke, "Towards a bayesian network model for predicting flaky automated tests," in *2018 IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C)*. IEEE, 2018, pp. 100–107.

[10] M. A. Mascheroni and E. Irrazbal, "Identifying key success factors in stopping flaky tests in automated rest service testing," 2018. [Online]. Available: http://portal.amelica.org/ameli/jatsRepo/30/308007/html/index.html

[11] H. Jiang, X. Li, Z. Yang, and J. Xuan, "What causes my test alarm? automatic cause analysis for test alarms in system and integration testing," in *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, 2017, pp. 712–723.

[12] G. Pinto, B. Miranda, S. Dissanayake, M. d'Amorim, C. Treude, and A. Bertolino, "What is the vocabulary of flaky tests?" in *MSR '20: 17th Int'l. Conf. on Mining Software Repositories*, 2020.

[13] A. Bertolino, E. Cruciani, B. Miranda, and R. Verdecchia, "Know your neighbor: fast static prediction of test flakiness (github)," https://github.com/ICSE2020-FLAST/FLAST, 2019.

[14] M. Harman and P. O'Hearn, "From start-ups to scale-ups: Opportunities and open problems for static and dynamic program analysis," in *2018 IEEE 18th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE, 2018, pp. 1–23.

[15] M. Waterloo, S. Person, and S. Elbaum, "Test analysis: Searching for faults in tests," in *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '15. IEEE Press, 2015, pp. 149–154. [Online]. Available: https://doi.org/10.1109/ASE.2015.37

[16] A. Shi, A. Gyori, O. Legunsen, and D. Marinov, "Detecting assumptions on deterministic implementations of non-deterministic specifications," in *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 2016, pp. 80–90.

[17] W. Lam, P. Godefroid, S. Nath, A. Santhiar, and S. Thummalapenta, "Root causing flaky tests in a large-scale industrial setting," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2019. New York, NY, USA: Association for Computing Machinery, 2019, pp. 101–111. [Online]. Available: https://doi.org/10.1145/3293882.3330570

[18] K. Herzig, "Let's assume we had to pay for testing," https://www.slideshare.net/slideshow/embed_code/63509323, 2016.

[19] A. Bertolino, E. Cruciani, B. Miranda, and R. Verdecchia, "Know your neighbor: fast static prediction of test flakiness," https://dx.doi.org/10.32079/ISTI-TR-2020/001, ISTI Technical Reports, Tech. Rep., 2020.

[20] A. Alshammari, C. Morris, M. Hilton, and J. Bell, "Flakeflagger," https://github.com/AlshammariA/FlakeFlagger, 2021.

[21] ——, "Flaky Test Dataset to Accompany 'FlakeFlagger: Predicting Flakiness Without Rerunning Tests'," Jan. 2021. [Online]. Available: https://doi.org/10.5281/zenodo.4450723

[22] W. Lam, S. Winter, A. Astorga, V. Stodden, and D. Marinov, "Understanding reproducibility and characteristics of flaky tests through test reruns in java projects," in *2020 IEEE 31st International Symposium on Software Reliability Engineering (ISSRE)*, 2020, pp. 403–413.

[23] S. Zhang, D. Jalali, J. Wuttke, K. Muşlu, W. Lam, M. D. Ernst, and D. Notkin, "Empirically revisiting the test independence assumption," in *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ser. ISSTA 2014. New York, NY, USA: Association for Computing Machinery, 2014, pp. 385–396. [Online]. Available: https://doi.org/10.1145/2610384.2610404

[24] A. Van Deursen, L. Moonen, A. Van Den Bergh, and G. Kok, "Refactoring test code," in *Proceedings of the 2nd international conference on extreme programming and flexible processes in software engineering (XP2001)*, 2001, pp. 92–95.

[25] G. Meszaros, *xUnit test patterns: Refactoring test code*. Pearson Education, 2007.

[26] V. Garousi and B. Küçük, "Smells in software test code: A survey of knowledge in industry and academia," *Journal of systems and software*, vol. 138, pp. 52–81, 2018.

[27] W. Lam, S. Winter, A. Wei, T. Xie, D. Marinov, and J. Bell, "A large-scale longitudinal study of flaky tests," *Proc. ACM Program. Lang.*, vol. 4, no. OOPSLA, Nov. 2020. [Online]. Available: https://doi.org/10.1145/3428270

[28] R. Ramler, M. Moser, and J. Pichler, "Automated static analysis of unit test code," in *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, vol. 2, March 2016, pp. 25–28.

[29] M. Breugelmans and B. V. Rompaey, "Testq: Exploring structural and maintenance characteristics of unit test suites," in *IN WASDETT-1*, 2008.

[30] M. Greiler, A. van Deursen, and M. Storey, "Automated detection of test fixture strategies and smells," in *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*. Los Alamitos, CA, USA: IEEE Computer Society, mar 2013, pp. 322–331. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/ICST.2013.45

[31] F. Palomba, A. Zaidman, and A. D. Lucia, "Automatic test smell detection using information retrieval techniques," in *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2018.

[32] G. Bavota, A. Qusef, R. Oliveto, A. Lucia, and D. Binkley, "Are test smells really harmful? an empirical study," *Empirical Softw. Engg.*, vol. 20, no. 4, pp. 1052–1094, Aug. 2015. [Online]. Available: https://doi.org/10.1007/s10664-014-9313-0

[33] A. Singh, N. Thakur, and A. Sharma, "A review of supervised machine learning algorithms," in *2016 3rd International Conference on Computing for Sustainable Global Development (INDIACom)*, March 2016, pp. 1310–1315.

[34] J. Dougherty, R. Kohavi, and M. Sahami, "Supervised and unsupervised discretization of continuous features," in *Machine Learning Proceedings 1995*. Elsevier, 1995, pp. 194–202.

[35] S. Kotsiantis and D. Kanellopoulos, "Discretization techniques: A recent survey," *GESTS International Transactions on Computer Science and Engineering*, vol. 32, no. 1, pp. 47–58, 2006.

[36] SciKit-Learn Developers, "scikit," 2020. [Online]. Available: https://scikit-learn.org/stable/

[37] S. Lei, "A feature selection method based on information gain and genetic algorithm," in *2012 International Conference on Computer Science and Electronics Engineering*, vol. 2. IEEE, 2012, pp. 355–358.

[38] R. Kohavi *et al.*, "A study of cross-validation and bootstrap for accuracy estimation and model selection," in *Ijcai*, vol. 14, no. 2. Montreal, Canada, 1995, pp. 1137–1145.

[39] Y. Bengio and Y. Grandvalet, "No unbiased estimator of the variance of k-fold cross-validation," *Journal of machine learning research*, vol. 5, no. Sep, pp. 1089–1105, 2004.

[40] N. V. Chawla, K. W. Bowyer, L. O. Hall, and W. P. Kegelmeyer, "Smote: synthetic minority over-sampling technique," *Journal of artificial intelligence research*, vol. 16, pp. 321–357, 2002.

[41] N. V. Chawla, "Data mining for imbalanced datasets: An overview," in *Data mining and knowledge discovery handbook*. Springer, 2009, pp. 875–886.

[42] "Where do our flaky tests come from?" Apr 2017. [Online]. Available: https://testing.googleblog.com/2017/04/where-do-our-flaky-tests-come-from.html

[43] W. Lam, "Illinois Dataset of Flaky Tests (IDoFT)," 2020. [Online]. Available: http://mir.cs.illinois.edu/flakytests

[44] T. Winters, T. Manshreck, and H. Wright, *Software Engineering at Google: Lessons Learned from Programming Over Time*. O'Reilly Media, 2020. [Online]. Available: https://books.google.com/books?id=TyIrywEACAAJ

[45] A. Peruma, K. Almalki, C. D. Newman, M. W. Mkaouer, A. Ouni, and F. Palomba, "tsdetect: An open source test smells detection tool," in *Proceedings of the 2020 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2020. New York, NY, USA: Association for Computing Machinery, Nov. 2020.

[46] A. Gyori, A. Shi, F. Hariri, and D. Marinov, "Reliable testing: Detecting state-polluting tests to prevent test dependency," in *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ser. ISSTA 2015, 2015, pp. 223–233.

[47] J. Bell, G. Kaiser, E. Melski, and M. Dattatreya, "Efficient dependency detection for safe java test acceleration," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2015. New York, NY, USA: Association for Computing Machinery, 2015, pp. 770–781. [Online]. Available: https://doi.org/10.1145/2786805.2786823

[48] C. Huo and J. Clause, "Improving oracle quality by detecting brittle assertions and unused inputs in tests," in *International Symposium on Foundations of Software Engineering*, 2014, pp. 621–631.

[49] A. Shi, W. Lam, R. Oei, T. Xie, and D. Marinov, "ifixflakies: A framework for automatically fixing order-dependent flaky tests," in

[57] D. Di Nucci, F. Palomba, D. A. Tamburri, A. Serebrenik, and A. De Lucia, "Detecting code smells using machine learning techniques: are we there yet?" in *2018 IEEE 25th International Conference on Software Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, 2019, pp. 545–555.

[50] P. E. Strandberg, T. J. Ostrand, E. J. Weyuker, W. Afzal, and D. Sundmark, "Intermittently failing tests in the embedded systems domain," in *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2020. New York, NY, USA: Association for Computing Machinery, 2020, p. 337348. [Online]. Available: https://doi.org/10.1145/3395363.3397359

[51] A. Shi, J. Bell, and D. Marinov, "Mitigating the effects of flaky tests on mutation testing," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2019. New York, NY, USA: Association for Computing Machinery, 2019, pp. 112–122. [Online]. Available: https://doi.org/10.1145/3293882.3330568

[52] M. Cordy, R. Rwemalika, M. Papadakis, and M. Harman, "Flakime: Laboratory-controlled test flakiness impact assessment. a case study on mutation testing and program repair," *arXiv preprint arXiv:1912.03197*, 2019.

[53] T. Guggulothu and S. A. Moiz, "Code smell detection using multi-label classification approach," *Software Quality Journal*, 2020. [Online]. Available: https://doi.org/10.1007/s11219-020-09498-y

[54] M. I. Azeem, F. Palomba, L. Shi, and Q. Wang, "Machine learning techniques for code smell detection: A systematic literature review and meta-analysis," *Information and Software Technology*, 2019.

[55] F. A. Fontana, M. Zanoni, A. Marino, and M. V. Mäntylä, "Code smell detection: Towards a machine learning-based approach," in *2013 IEEE International Conference on Software Maintenance*. IEEE, 2013, pp. 396–399.

[56] R. Chang, S. Sankaranarayanan, G. Jiang, and F. Ivancic, "Software testing using machine learning," Dec. 30 2014, uS Patent 8,924,938.
*Analysis, Evolution and Reengineering (SANER)*. IEEE, 2018, pp. 612–621.

[58] V. H. Durelli, R. S. Durelli, S. S. Borges, A. T. Endo, M. M. Eler, D. R. Dias, and M. P. Guimarães, "Machine learning applied to software testing: A systematic mapping study," *IEEE Transactions on Reliability*, 2019.

[59] G. Catolino, F. Palomba, F. A. Fontana, A. De Lucia, A. Zaidman, and F. Ferrucci, "Improving change prediction models with code smell-related information," vol. 25, no. 1, 2020, pp. 49–95. [Online]. Available: https://doi.org/10.1007/s10664-019-09739-0