White Rose Research Online

eprints@whiterose.ac.uk
https://eprints.whiterose.ac.uk
Universities of Leeds, Sheffield and York

Deposited via The University of Sheffield.

White Rose Research Online URL for this paper:
https://eprints.whiterose.ac.uk/id/eprint/230092/

Version: Accepted Version

Proceedings Paper:
Parry, O., Kapfhammer, G.M., Hilton, M. et al. (2022) Evaluating features for machine learning detection of order- and non-order-dependent flaky tests. In: Proceedings of 2022 IEEE Conference on Software Testing, Verification and Validation (ICST). 2022 IEEE Conference on Software Testing, Verification and Validation (ICST), 04-14 Apr 2022, Valencia, Spain. Institute of Electrical and Electronics Engineers (IEEE), pp. 93-104. ISBN: 9781665466806. ISSN: 2159-4848. EISSN: 2771-3091.

https://doi.org/10.1109/icst53961.2022.00021

© 2022 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other users, including reprinting/ republishing this material for advertising or promotional purposes, creating new collective works for resale or redistribution to servers or lists, or reuse of any copyrighted components of this work in other works. Reproduced in accordance with the publisher's self-archiving policy.

Reuse
Items deposited in White Rose Research Online are protected by copyright, with all rights reserved unless indicated otherwise. They may be downloaded and/or printed for private study, or other acts as permitted by national copyright laws. The publisher or other rights holders may allow further reproduction and re-use of the full text version. This is indicated by the licence information on the White Rose Research Online record for the item.

Takedown
If you consider content in White Rose Research Online to be in breach of UK law, please notify us by emailing eprints@whiterose.ac.uk including the URL of the record and the reason for the withdrawal request.

UNIVERSITY OF LEEDS
University of Sheffield
UNIVERSITY of York

Evaluating Features for Machine Learning Detection of Order- and Non-Order-Dependent Flaky Tests

Owain Parry
University of Sheffield

Gregory M. Kapfhammer
Allegheny College

Michael Hilton
Carnegie Mellon University

Phil McMinn
University of Sheffield

***Abstract*—Flaky tests are test cases that can pass or fail without code changes. They often waste the time of software developers and obstruct the use of continuous integration. Previous work has presented several automated techniques for detecting flaky tests, though many involve repeated test executions and a lot of source code instrumentation and thus may be both intrusive and expensive. While this motivates researchers to evaluate machine learning models for detecting flaky tests, prior work on the features used to encode a test case is limited. Without further study of this topic, machine learning models cannot perform to their full potential in this domain. Previous studies also exclude a specific, yet prevalent and problematic, category of flaky tests: order-dependent (OD) flaky tests. This means that prior research only addresses part of the challenge of detecting flaky tests with machine learning. Closing this knowledge gap, this paper presents a new feature set for encoding tests, called FLAKE16. Using 54 distinct pipelines of data preprocessing, data balancing, and machine learning models for detecting both non-order-dependent (NOD) and OD flaky tests, this paper compares FLAKE16 to another well-established feature set. To assess the new feature set's effectiveness, this paper's experiments use the test suites of 26 Python projects, consisting of over 67,000 tests. Along with identifying the most impactful metrics for using machine learning to detect both types of flaky test, the empirical study shows how FLAKE16 is better than prior work, including (1) a 13% increase in overall F1 score when detecting NOD flaky tests and (2) a 17% increase in overall F1 score when detecting OD flaky tests.**

***Index Terms*—Software Testing, Flaky Tests, Machine Learning**

I. INTRODUCTION

Flaky tests are test cases that can both pass or fail without any changes to the test code or the code under test and are therefore an unreliable indicator of software correctness [38]. They are a significant problem in software development because they may lead to time wasted investigating a non-existent bug, or potentially more seriously, might mask the presence of a genuine bug [42], [61], [66]. Their non-deterministic behavior also hinders continuous integration: a study focused on the Travis CI platform found that 47% of failing builds that were manually restarted eventually passed without any changes, indicating the presence of flaky tests [16]. As well as for open-source development, flaky tests are a major problem for well-known software companies, such as Google, Microsoft, and Facebook [30], [39], [40]. A survey of software developers found that 79% of respondents considered flaky tests to be a moderate or serious problem, with 59% encountering them on a monthly, weekly, or daily basis [18]. A specific category of flaky tests, known as *order-dependent* (OD) flaky tests, are influenced by previously executed test cases. Previous studies have found these tests to be a very prevalent type of flaky test [32], [34]. Their order-based non-determinism makes them a major obstacle to the application of techniques that aim to gather useful results from testing sooner, such as test case prioritization, selection, and parallelization [6], [11], [33].

Given the problems associated with flaky tests, the research community has developed automated techniques to detect them. Many of these techniques involve a significant number of repeated test executions and some require extensive instrumentation [6], [9], [17], [19], [32], [57], [66], making them prohibitively expensive for practical deployment in large software projects. This motivated researchers to develop detection techniques based on machine learning models that they trained using static features of test cases, such as their length, complexity, and the presence of particular keywords and identifiers [8], [44]. For instance, one study found that combining static features with several dynamically-collected characteristics, like execution time and line coverage, resulted in significantly better detection performance at the relatively minimal cost of a single, instrumented test suite run [3].

To date, prior studies have only evaluated a limited range of features, while the broader literature has identified many more test case characteristics that may be indicative of flakiness. Without further evaluation of a wider range of features, machine learning models cannot be used to their full potential for detecting flaky tests. Moreover, previous studies trained and evaluated models using datasets of flaky tests that do not include OD flaky tests [7]. Yet, Lam et al. found that over 60% of their detected flaky tests were OD [32], suggesting that previous studies may have labelled a large portion of flaky tests as *non-flaky* when training models. This means they have only considered a subset of the problem of flaky test detection. Given the aforementioned difficulties caused by OD flaky tests, their efficient detection has significant benefits [6], [11], [33].

This paper's study evaluates the performance of 54 pipelines of data preprocessing, data balancing, and machine learning models for detecting flaky tests in 26 open-source Python projects. Given previous successes with the *random forest* model [3], [44], [56], it focuses on the *decision tree* model and ensemble models thereof [20], [51]. It also introduces FLAKE16, a new feature set for encoding test cases using seven metrics from a previously established feature set [3] and nine additional metrics, including the depth of the abstract syntax tree of the test's code and the maximum memory usage during test case execution. The results show that FLAKE16

TABLE I

THE METRICS OF FLAKE16. THE "FF" COLUMN INDICATES IF THE FEATURE IS ALSO PART OF THE FLAKEFLAGGER FEATURE SET. THE "STATIC" COLUMN INDICATES IF THE FEATURE CAN BE MEASURED WITHOUT EXECUTING THE TEST CASE. THE "IMPACT RANK" COLUMNS ARE THE RANKS OF EACH FEATURE IN DESCENDING ORDER OF IMPACTFULNESS FOR DETECTING BOTH NOD AND OD FLAKY TESTS (SEE FIGURE 2 FOR MORE DETAILS).

					Impact Rank	
#	Feature	Description	FF	Static	NOD	OD
1	Covered Lines	Number of lines covered.	✓		8	6
2	Covered Changes	Total number of times each covered line has been modified in the last 75 commits.	✓		3	5
3	Source Covered Lines	Number of lines covered that are not part of test cases.	✓		7	7
4	Execution Time	Elapsed wall-clock time of the test case execution.	✓		5	9
5	Read Count	Number of times the filesystem had to perform input [28].			6	2
6	Write Count	Number of times the filesystem had to perform output [28].			4	1
7	Context Switches	Number of voluntary context switches.			10	8
8	Max. Threads	Peak number of concurrently running threads (excluding the main thread).			1	11
9	Max. Memory	Peak memory usage.			11	4
10	AST Depth	Maximum depth of nested program statements in the test case code.		✓	2	13
11	Assertions	Number of assertion statements in the test case code.	✓	✓	14	3
12	External Modules	Number of non-standard modules (i.e., libraries) used by the test case.		✓	16	16
13	Halstead Volume	A measure of the size of an algorithm's implementation [2], [43], [45].		✓	15	14
14	Cyclomatic Complexity	Number of branches in the test case code [21], [43], [45].		✓	12	12
15	Test Lines of Code	Number of lines in the test case code [43], [45].	✓	✓	9	10
16	Maintainability	A measure of how easy the test case code is to support and modify [48], [64].		✓	13	15

offered a 13% increase in overall F1 score compared to the previous feature set when detecting non-order-dependent (NOD) flaky tests. The experiments also study the same machine learning pipelines with both feature sets for the task of detecting OD flaky tests. In this setup, FLAKE16 offered a 17% increase in the overall F1 score. Finally, the paper studies the impact of each FLAKE16 feature on the models' predictions, revealing that the peak number of concurrently running threads and the number of read- and write-related system calls during test execution are the most valuable features for detecting NOD and OD flaky tests, respectively.

In summary, the main contributions of this paper are:

Contribution 1: New Feature Set. The paper introduces FLAKE16, a new feature set for machine learning-based flaky test detection. The evaluation demonstrates an improved detection performance for both NOD and OD flaky tests compared to a previous feature set, as further detailed in Section II.

Contribution 2: Novel Evaluation. Our evaluation of 54 machine learning pipelines is the first to consider the detection of OD flaky tests, offering a more complete assessment of the applicability of machine learning to the problem of flaky test detection. See Section III for more details on this contribution.

Contribution 3: Findings and Implications. Leveraging the empirical results, the paper surfaces findings with implications relevant to both the research community and software developers, including the most impactful test case metrics for detecting flaky tests. See Sections IV and V for details on these findings.

Contribution 4: Framework and Data. To collect the data required to train the machine learning pipelines and perform the experiments, we developed our own comprehensive framework of tools, called FLAKE16FRAMEWORK. To identify flaky tests, we used the FLAKE16FRAMEWORK to execute 5,000 times the test suites for 26 programs containing 67,000 test cases in total. Supporting the replication of this paper's results and further investigations into the use of machine learning for flaky test detection, we make FLAKE16FRAMEWORK and all of our data available as part of our replication package [49].

II. THE FLAKE16 FEATURE SET

Alshammari et al. [3] proposed a range of features for encoding test cases in machine learning-based flaky test detection and split them into two groups. These were eight boolean features indicating the presence of test smells [60], and eight numerical features measuring a mixture of static and dynamic test case properties. They found the test smell features to be of limited value and excluded them from their evaluation of their flaky test detection framework, FLAKEFLAGGER. One of the remaining eight features, the total number of production classes covered by a test case, was not applicable in the context of our study. This is because the dataset of test cases used by Alshammari et al. are from Java projects [7] and ours are from Python projects. In Java, classes are a central construct for building programs, whereas in Python, they are less critical and it is possible to write programs without them [13]. We refer to the remaining seven features as the FLAKEFLAGGER feature set, which is subsumed by FLAKE16. One of these features captures the "churn" of the lines covered by a test case, that is, how frequently they are changed. This requires a window of past commits to consider. Alshammari et al. evaluated eight windows and found 75 commits to be the most informative, and thus we selected this value for this paper's study. Beyond these seven features, FLAKE16 contains nine more static and dynamic test case metrics, with Table I providing a summary.

Several empirical studies identified files as a potential vector for OD flaky tests to arise [6], [9], [19], [38], [66]. In particular, Zhang et al. [66] found that 39% of OD flaky tests were caused by side effects left behind by other test cases in external resources, such as files and databases. Furthermore, flaky tests specifically caused by complications during input

and output operations were one of the flaky test categories presented by Luo et al. [38]. This motivated our inclusion of *read count* and *write count* in FLAKE16. Specifically, these measure the number of read- and write-related system calls during test execution. Another finding that many empirical studies have in common is that asynchronous operations and concurrency are very frequent causes of flaky tests [18], [31], [38], [50]. For this reason, we incorporated *context switches* and *maximum threads* into FLAKE16. The former measures the number of *voluntary* context switches performed during test case execution. These occur when a process gives up its CPU time because it has nothing to do, which would occur when a test case sleeps for a fixed amount of time. Previous studies have identified this as a hallmark of flaky tests in the asynchronous category [18], [38]. We also integrated *maximum memory* into FLAKE16. This feature measures the peak memory usage during test case execution, a property identified by an author of the Google Testing Blog to be correlated with the likelihood of a test case being flaky [35].

The FLAKE16 feature set also contains four additional static metrics that aim to capture the size and complexity of the test case code. A recent study identified this general property to be a possible indicator of flaky tests [45]. With that said, another recent study cast doubt on the reliability of various code complexity metrics for measuring program comprehension difficulty [43]. Nevertheless, this does not necessarily imply that they would be of no use for detecting flaky tests, so this paper evaluates them. The first of these is *Abstract Syntax Tree (AST) depth*. Specifically, this feature measures the maximum depth of nested program statements, such as `if` statements and `for` loops. The second is *Halstead volume*, which attempts to capture the "size" of an algorithm's implementation. Where N is the *total* number of operators and operands in the test case code and η is the number of *distinct* operators and operands, Halstead volume is given by $Nlog_2(\eta)$ [2]. The third static metric is *cyclomatic complexity*, which measures the number of branches in a piece of code [21]. In Python, and many other programming languages, an `if` statement corresponds to a branch and so would increase the cyclomatic complexity by 1. Other examples of branches include `for` and `while` statements, since they both evaluate a condition before every iteration. The fourth metric is *maintainability*. This is an empirical measure of how easy a piece of code is to support and modify [64]. There are several formulations, though we used the one implemented by the RADON library [48]. We selected this library because it also contains implementations for calculating Halstead volume and cyclomatic complexity.

III. EVALUATION

We designed and conducted experiments to answer the following three research questions regarding the benefit of features during machine learning-based flaky test detection:

RQ1. Compared to the features used by FLAKEFLAGGER, does the FLAKE16 feature set improve the performance of flaky test case detection with machine learning models?

TABLE II
THE 26 OPEN-SOURCE PYTHON PROJECTS EXAMINED IN THIS PAPER'S STUDY. THE "STARS" COLUMN IS THE NUMBER OF TIMES A GITHUB USER HAS INDICATED THEIR INTEREST IN THE PROJECT [52]. THE "TESTS" COLUMN IS THE TOTAL NUMBER OF TEST CASES, BOTH FLAKY AND NON-FLAKY. THE "NOD" AND "OD" COLUMNS ARE THE NUMBER OF NON-ORDER-DEPENDENT AND ORDER-DEPENDENT FLAKY TESTS.

GitHub Repository	# Stars	# Tests	# NOD	# OD
apache/airflow	23175	3458	66	293
celery/celery	17952	2365	-	15
conan-io/conan	5274	3707	-	13
encode/django-rest-framework	21906	1402	-	1
spesmilo/electrum	5154	544	1	1
Flexget/Flexget	1342	1335	1	4
fonttools/fonttools	2850	3456	1	42
graphql-python/graphene	6810	347	-	1
facebookresearch/hydra	4861	1540	-	19
HypothesisWorks/hypothesis	5379	4386	5	6
ipython/ipython	14982	846	6	304
celery/kombu	2221	1025	2	23
apache/libcloud	1788	9840	3	133
Delgan/loguru	9838	1255	4	21
mitmproxy/mitmproxy	24702	1231	-	17
python-pillow/Pillow	8983	2583	-	26
PrefectHQ/prefect	6897	7038	25	20
PyGithub/PyGithub	4664	711	-	4
Pylons/pyramid	3593	2633	-	4
psf/requests	46050	537	5	-
scikit-image/scikit-image	4525	6281	-	12
mwaskom/seaborn	8772	1028	1	8
pypa/setuptools	1439	704	1	23
sunpy/sunpy	629	2072	-	2
urllib3/urllib3	2788	1900	15	1
xonsh/xonsh	5133	4782	9	19
Total	241707	67006	145	1012

RQ2. Can machine learning models be applied to effectively detect order-dependent flaky test cases?

RQ3. Which features of FLAKE16 are the most impactful?

A. Data Collection

To evaluate the performance of any machine learning model for detecting flaky tests, we needed a labelled dataset of test cases. To that end, we sampled 26 popular Python projects, most of which are considered critical to open-source infrastructure [41]. In total, these 26 projects, listed in Table II, gave us a dataset of over 67,000 test cases. In order to train and evaluate a machine learning classifier, we needed to label each test case as *non-flaky*, *NOD flaky*, or *OD flaky*. To that end, we created a framework of tools, called FLAKE16FRAMEWORK, to automatically execute each project's test suite 2,500 times in a consistent order and an additional 2,500 times in a random order. For reproducibility and isolation between test suite runs, FLAKE16FRAMEWORK installs each project inside of a fully-specified virtual environment [63] to produce a Docker image [15], which it uses to create a separate container for each test suite run. The framework also records the outcome (i.e., pass or fail) of every test during each test suite execution. It labels a test as *NOD flaky* if it has an inconsistent outcome during the runs in a consistent order. Otherwise, it labels a test as *OD flaky* if it has an inconsistent outcome during the runs in

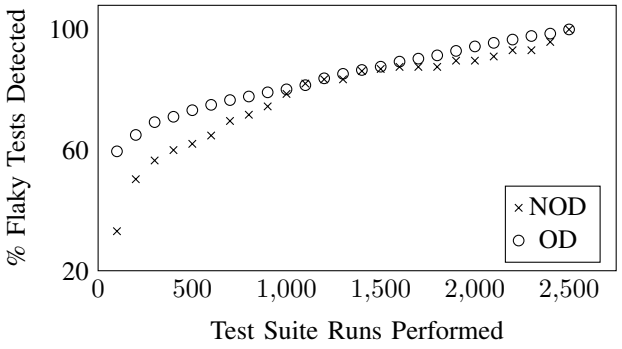

Fig. 1. The relationship between the number of test suites runs performed by FLAKE16FRAMEWORK and the percentage of flaky tests it identified, both NOD and OD. As the curves show, the relationship in both cases is sublinear.

random orders. Failing that, it labels a test as *non-flaky*. This is an established practice for identifying flaky tests [23], [34].

Given the non-deterministic nature of flaky tests, it is impossible to label a test case as non-flaky with complete certainty. Naturally, confidence increases with the number of test suite runs, but so too does the computational cost. Alshammari et al. [3] executed test suites 10,000 times. Based on their findings, the cumulative number of detected flaky tests appears sublinearly related to the number of test suite runs. In other words, continuing to re-execute a test suite gives diminishing returns with respect to the confidence of labelling a test case as non-flaky. Our study confirms these findings, for both NOD and OD flaky tests, as illustrated by the curves in Figure 1. As such, we selected a smaller number of test suite runs to reduce the time to finish the labelling process. Despite this, labelling still took over four weeks of computational time on a computer with a 24-core AMD Ryzen 5900X CPU.

As well as having labels for each test case, we also needed to measure values for each of the metrics of FLAKE16. To that end, we designed FLAKE16FRAMEWORK to perform the necessary static analysis on the source code of every test case and to instrument test case execution to collect the dynamic features. We implemented this with the help of several existing Python libraries. To collect most of the static metrics, we used the RADON library [48]. To determine the number of external modules used by a test case, we implemented our own approach that analyzes the AST of a test case. To measure line coverage data, we used COVERAGE.PY [14]. For the majority of the remaining dynamic features, we used PSUTIL [47]. In keeping with previous work [3], FLAKE16FRAMEWORK executed each of the 26 test suites just once to measure these values to keep its computational cost as low as possible. While there may be some expected variance in these values, we leave it as future work to investigate if the repeated measurement of these features improves the performance of flaky test detection.

B. Data Preprocessing

Preprocessing of raw feature data is a typical component of machine learning pipelines [22], [65]. To that end, we evaluated two common data preprocessing techniques. The first was *scaling* (also known as *standardization*), which, for each feature, involves subtracting the mean over the entire dataset and dividing by the standard deviation. This has the effect of "centering" the distribution of each feature with a mean of zero and a variance of one, such as a standard normal distribution. This is a common requirement for many machine learning models [46]. The second was *principal component analysis* (PCA) [1]. This is a technique used to transform a dataset such that each new feature corresponds to a *principal component*. The principal components of a dataset can be thought of as an ordered set of orthogonal vectors representing axes that best capture the variance of the data. The first principal component captures the most variance and the subsequent components capture increasingly less. A common use of PCA is to reduce the number of features in a dataset while sacrificing as little data as possible. This is known as *dimensionality reduction* [62]. Because the principal components are orthogonal to one another, PCA also decorrelates the features of a dataset. Since the dimensionality of the dataset is relatively low, and Table III shows that many of FLAKE16's features are correlated, decorrelation is our primary use case for PCA.

C. Data Balancing

As shown by Table II, the number of non-flaky tests in our dataset vastly outnumbers both the NOD and OD flaky tests. Training machine learning models with imbalanced data such as ours potentially limits their performance [36], [58]. To address this, we evaluated five data balancing techniques. Data balancing techniques can be split into two categories: those that *undersample* (reduce) the majority class (non-flaky), and those that *oversample* (increase) the minority class (flaky). We evaluated two undersampling, one oversampling, and two combined techniques. The first undersampling technique removes the non-flaky samples within *Tomek links* of the dataset. A Tomek link occurs between two samples when a sample of one class is the nearest neighbor of a sample of the other [59]. The second technique, *edited nearest-neighbors*, removes non-flaky samples whose nearest neighbors are all flaky. With a neighborhood of only one sample, edited nearest-neighbors is equivalent to the previous technique. We used a neighborhood of three, our implementation's default. For oversampling, we evaluated the *synthetic minority oversampling technique* (SMOTE) [12], which generates synthetic flaky samples by interpolation. The two combined techniques we evaluated were the combination of SMOTE with Tomek links and edited nearest-neighbors. In both instances, SMOTE is applied first and the undersampling technique acts as a data cleaning method, rather than to undersample the non-flaky samples [5]. To ensure the correctness of these balancing techniques, we used those in the IMBALANCED-LEARN Python library [27].

D. Machine Learning Models

For the classification of test cases as flaky or non-flaky, we evaluated three machine learning models. Previous studies have found the *random forest* model [10] to be particularly performant in this domain [3], [44]. Random forest is an *ensemble* model, which combines many *base models*, in this

TABLE III

SPEARMAN RANK-ORDER CORRELATION COEFFICIENTS BETWEEN EACH PAIR OF FEATURES IN FLAKE16. VALUES RANGE BETWEEN -1 AND 1, WITH 0 INDICATING NO CORRELATION AND -1 OR 1 INDICATING AN EXACT MONOTONIC RELATIONSHIP. A NEGATIVE VALUE INDICATES THAT AS THE FEATURE IN THE ROW INCREASES, THE FEATURE IN THE COLUMN DECREASES. DARKER SHADED CELLS INDICATE A GREATER MAGNITUDE OF CORRELATION.

	Covered Lines	*Covered Changes*	*Source Covered Lines*	*Execution Time*	*Read Count*	*Write Count*	*Context Switches*	*Max. Threads*	*Max. Memory*	*AST Depth*	*Assertions*	*External Modules*	*Halstead Volume*	*Cyclomatic Complexity*	*Test Lines of Code*	*Maintainability*
Covered Lines	1.00	0.72	1.00	0.48	-0.05	0.32	0.50	0.15	0.21	-0.09	-0.07	-0.30	-0.11	-0.08	0.15	0.09
Covered Changes	0.72	1.00	0.72	0.41	-0.01	0.25	0.35	0.13	0.21	0.04	0.14	-0.14	0.08	0.13	0.17	-0.09
Source Covered Lines	1.00	0.72	1.00	0.48	-0.05	0.32	0.49	0.14	0.21	-0.10	-0.06	-0.31	-0.10	-0.07	0.13	0.08
Execution Time	0.48	0.41	0.48	1.00	0.55	0.48	0.48	0.30	0.76	0.03	0.17	-0.01	0.13	0.14	0.09	-0.12
Read Count	-0.05	-0.01	-0.05	0.55	1.00	0.40	0.12	0.18	0.73	0.17	0.21	0.30	0.21	0.22	0.12	-0.18
Write Count	0.32	0.25	0.32	0.48	0.40	1.00	0.52	0.32	0.45	0.00	-0.07	-0.06	-0.04	-0.07	0.13	0.04
Context Switches	0.50	0.35	0.49	0.48	0.12	0.52	1.00	0.31	0.36	-0.11	-0.04	-0.18	-0.05	-0.06	0.09	0.07
Max. Threads	0.15	0.13	0.14	0.30	0.18	0.32	0.31	1.00	0.26	0.05	-0.01	0.05	0.01	0.00	0.10	-0.01
Max. Memory	0.21	0.21	0.21	0.76	0.73	0.45	0.36	0.26	1.00	0.10	0.21	0.16	0.20	0.20	0.06	-0.16
AST Depth	-0.09	0.04	-0.10	0.03	0.17	0.00	-0.11	0.05	0.10	1.00	0.21	0.16	0.24	0.42	0.42	-0.26
Assertions	-0.07	0.14	-0.06	0.17	0.21	-0.07	-0.04	-0.01	0.21	0.21	1.00	0.15	0.73	0.89	0.30	-0.69
External Modules	-0.30	-0.14	-0.31	-0.01	0.30	-0.06	-0.18	0.05	0.16	0.16	0.15	1.00	0.28	0.18	0.18	-0.24
Halstead Volume	-0.11	0.08	-0.10	0.13	0.21	-0.04	-0.05	0.01	0.20	0.24	0.73	0.28	1.00	0.75	0.35	-0.91
Cyclomatic Complexity	-0.08	0.13	-0.07	0.14	0.22	-0.07	-0.06	0.00	0.20	0.42	0.89	0.18	0.75	1.00	0.43	-0.72
Test Lines of Code	0.15	0.17	0.13	0.09	0.12	0.13	0.09	0.10	0.06	0.42	0.30	0.18	0.35	0.43	1.00	-0.39
Maintainability	0.09	-0.09	0.08	-0.12	-0.18	0.04	0.07	-0.01	-0.16	-0.26	-0.69	-0.24	-0.91	-0.72	-0.39	1.00

case *decision tree* [51]. Decision tree is a non-parametric model that learns simple *if-then-else* decision rules from the training data, forming a binary tree. In this context, their output is the estimated probability of a test case being non-flaky. The random forest model trains each decision tree on a *random sample with replacement* of the training data, that is, a sample where individual data points can appear more than once. The overall classification is based on the average of their estimated probabilities. A related model, *extremely randomized trees* [20], also known as *extra trees*, trains trees on random samples *without* replacement and introduces additional randomization in how they are trained. We evaluated random forest and extra trees, as well as the base decision tree model, using the implementations provided by SCIKIT-LEARN [53].

E. Methodology

We used the FLAKE16FRAMEWORK to evaluate all combinations of data preprocessing, data balancing, and machine learning model, including no preprocessing and no balancing. This resulted in 54 machine learning pipelines. The framework evaluated these using both the FLAKE16 and the FLAKEFLAGGER feature sets and applied them to two binary classification problems, *non-flaky or NOD flaky* and *non-flaky or OD flaky*, culminating in 216 models. It used *stratified 10-fold cross validation* for model training and testing, as done by Alshammari et al. in their evaluation of their FLAKEFLAGGER framework [3]. This creates 10 *folds*, in which 90% of the dataset is used for training and 10% is used for testing. The class balance of each fold roughly follows that of the entire dataset, and FLAKE16FRAMEWORK applied data balancing to the training set only. This is so the model testing accurately reflects the imbalanced nature of the classification problem. After training the model, the framework applied it to each test case of the testing set, resulting in a prediction of flaky or non-flaky. Since the testing portion of each fold is unique, after 10 folds every test case in the dataset has a predicted label.

Where flaky is the *positive* label and a predicted label for a test case is *true* if it matches the label assigned to it during data collection, FLAKE16FRAMEWORK enumerated the number of *false positives*, *false negatives*, and *true positives*, for the test cases of each project and for the entire dataset. From these, it calculated the *precision*, the ratio of true positives to all positives, and the *recall*, the ratio of true positives to the sum of true positives and false negatives. In other words, precision is the fraction of genuine flaky tests among all test cases the model labelled as flaky and recall is the fraction of genuine flaky tests that the model labelled as flaky. The framework also calculated the *F1 score*, or the harmonic mean of these two metrics. These are all common metrics used in previous studies of machine learning to detect flaky tests [3], [8], [44]. The F1 score metric is particularly well-suited to imbalanced binary classification problems, like the one in this paper, since it penalizes significant differences between a model's precision and recall. This is important because a model that trivially labels all test cases as non-flaky would achieve maximum recall but very poor precision. We answered **RQ1** by comparing the performance of the best pipeline using the FLAKEFLAGGER feature set to the best using the FLAKE16 set, for both the NOD and OD classification problems. We answered **RQ2** by specifically focusing on the OD classification problem.

TABLE IV
THE TOP 10 PIPELINES WITH BOTH FEATURE SETS FOR DETECTING BOTH NOD AND OD FLAKY TESTS. TRAIN TIME AND TEST TIME ARE IN SECONDS.

FLAKEFLAGGER						FLAKE16					
Pre.	**Balancing**	**Model**	**Train Time**	**Test Time**	**F1**	**Pre.**	**Balancing**	**Model**	**Train Time**	**Test Time**	**F1**
NOD											
None	Tomek Links	Extra Trees	8.41	0.27	0.46	PCA	SMOTE	Extra Trees	133.72	0.57	0.52
Scaling	None	Extra Trees	11.73	0.37	0.44	PCA	SMOTE Tomek	Extra Trees	100.77	0.26	0.52
None	SMOTE Tomek	Extra Trees	41.17	0.26	0.43	Scaling	SMOTE Tomek	Extra Trees	123.84	0.59	0.51
None	SMOTE	Extra Trees	21.12	0.21	0.43	Scaling	SMOTE	Extra Trees	139.19	0.49	0.51
None	ENN	Extra Trees	6.10	0.13	0.43	None	SMOTE	Random Forest	269.69	0.22	0.48
None	None	Extra Trees	14.43	0.49	0.43	None	ENN	Extra Trees	34.94	0.25	0.48
Scaling	ENN	Extra Trees	10.58	0.46	0.43	PCA	SMOTE ENN	Extra Trees	146.26	0.53	0.48
Scaling	Tomek Links	Extra Trees	4.78	0.16	0.43	Scaling	SMOTE Tomek	Random Forest	209.72	0.25	0.48
Scaling	SMOTE Tomek	Extra Trees	41.20	0.56	0.42	None	SMOTE Tomek	Extra Trees	53.42	0.20	0.48
PCA	Tomek Links	Extra Trees	13.11	0.27	0.42	PCA	SMOTE Tomek	Random Forest	334.66	0.32	0.48
OD											
None	SMOTE Tomek	Extra Trees	70.29	1.23	0.47	Scaling	SMOTE	Random Forest	148.34	0.52	0.55
None	SMOTE	Extra Trees	75.57	1.16	0.46	Scaling	SMOTE Tomek	Random Forest	173.06	0.65	0.55
None	SMOTE Tomek	Random Forest	83.82	0.81	0.46	Scaling	SMOTE	Extra Trees	150.09	0.92	0.54
None	SMOTE	Random Forest	85.06	0.28	0.45	Scaling	SMOTE Tomek	Extra Trees	155.38	1.49	0.54
None	ENN	Extra Trees	13.75	0.80	0.45	None	SMOTE	Extra Trees	128.18	1.24	0.53
Scaling	ENN	Extra Trees	13.40	0.92	0.45	Scaling	ENN	Extra Trees	37.62	0.79	0.52
Scaling	Tomek Links	Extra Trees	18.03	1.08	0.45	None	SMOTE Tomek	Extra Trees	38.68	0.43	0.52
Scaling	None	Extra Trees	21.20	1.17	0.44	None	Tomek Links	Extra Trees	27.47	0.62	0.51
None	None	Extra Trees	12.92	0.69	0.44	Scaling	Tomek Links	Extra Trees	29.62	0.79	0.51
None	Tomek Links	Extra Trees	14.50	0.70	0.44	None	ENN	Extra Trees	27.93	0.80	0.50

To rank each feature of FLAKE16 in terms of its impact, we used the *Shapely Additive Explanations* (SHAP) technique [37]. This automated technique leverages concepts from game theory to quantify the contribution that a feature has on the output of a model. SHAP requires a dataset (i.e., a matrix where each row is a data point and each column represents a feature) and a trained model, and it returns a matrix of SHAP values in the same shape as the dataset. Each column of the SHAP values matrix corresponds to the impact that the respective feature had on the decision of the trained model for each data point. In our case, FLAKE16FRAMEWORK applied SHAP to estimate the contribution of each feature of FLAKE16 to a model's estimated probability of each test case being non-flaky. It did this for the best non-PCA pipelines for both test case classification problems when using FLAKE16.

Since the features of a PCA-transformed dataset correspond to a linear combination of the original features [1], it would be difficult to relate their impact back to the features of FLAKE16. To answer **RQ3**, FLAKE16FRAMEWORK quantified the impact of each feature for detecting both NOD and OD flaky tests by taking the mean absolute value of each column of the SHAP values matrices for both pipelines. A common alternative technique we could have used is to calculate the *permutation importance* of each feature. Given a trained model, this would involve shuffling the values of each feature across the dataset and measuring the impact this has on the performance of the model when applied to the shuffled dataset (e.g., the F1 score). Yet, this technique can give misleading results when features are correlated [26], as Table III shows is the case for the features used to predict whether or not a test is flaky.

F. Threats to Validity

This section considers the potential threats to the validity of this paper's evaluation and discusses how we mitigated them. First, during data collection the FLAKE16FRAMEWORK could have labelled some flaky tests as non-flaky. Given the non-deterministic nature of flaky tests, it is impossible to fully rectify this issue, although we mitigated it by performing as many test suite runs as possible within the limits of the computational resources available to us. Furthermore, some specific categories of flaky tests are unlikely to be manifested by rerunning alone [38], [54]. The only category we made special arrangements to detect were OD flaky tests; we consider other categories requiring additional means to identify out of the scope of this study. Second, FLAKE16FRAMEWORK could have contained bugs, which may have impacted the results of our evaluation. To that end, we used well-established Python libraries for the bulk of its functionality. These included COVERAGE.PY [14] to measure line coverage, PSUTIL [47] to measure many other dynamic properties of test cases, and SCIKIT-LEARN [53] for our model implementations. These are all popular open-source projects with many contributors, giving us confidence that any bugs would be identified, documented, and patched in a timely manner. We also wrote unit tests for greater confidence in the correctness of the bespoke elements of FLAKE16FRAMEWORK. Third, individual projects with significantly more test cases than others could bias the overall results. For example, AIRFLOW had the highest number of NOD flaky tests at 66, or 264% more than that of the second highest. To resolve this concern, we calculated performance metrics with respect to each individual project.

TABLE V
FOR NOD FLAKY TESTS, THE PER-PROJECT COMPARISON OF THE BEST PIPELINE FOR BOTH FEATURE SETS.

In this table, FP, FN, and TP are false positives, false negatives, and true positives, respectively. Finally, Pr stands for precision and Re is recall.

	FLAKEFLAGGER						FLAKE16					
Project	**FP**	**FN**	**TP**	**Pr**	**Re**	**F1**	**FP**	**FN**	**TP**	**Pr**	**Re**	**F1**
airflow	7	37	29	0.81	0.44	0.57	20	30	36	0.64	0.55	0.59
ipython	0	4	2	1.00	0.33	0.50	3	2	4	0.57	0.67	0.62
loguru	1	2	2	0.67	0.50	0.57	1	2	2	0.67	0.50	0.57
prefect	2	17	8	0.80	0.32	0.46	5	12	13	0.72	0.52	0.60
requests	1	3	2	0.67	0.40	0.50	1	3	2	0.67	0.40	0.50
xonsh	4	4	5	0.56	0.56	0.56	3	5	4	0.57	0.44	0.50
Total	16	97	48	0.75	0.33	0.46	49	76	69	0.58	0.48	0.52

IV. EMPIRICAL RESULTS

RQ1. Compared to the features used by FLAKEFLAGGER, does the FLAKE16 feature set improve the performance of flaky test case detection with machine learning models? The top half of Table IV shows the top 10 pipelines for detecting NOD flaky tests with both the FLAKEFLAGGER and FLAKE16 feature sets. The best pipeline with the FLAKE16 feature set was preprocessing with PCA, balancing with SMOTE, and extra trees as the model. Its F1 score was 13% higher than the best pipeline with the FLAKEFLAGGER feature set. Table V shows the per-project performance of the best pipelines with both feature sets. This table excludes projects for which we could not calculate precision, recall, or F1 score due to a division by zero. For all the included projects, with the exception of XONSH, the F1 score is either unchanged or higher with FLAKE16. Overall, the best pipeline with FLAKE16 had a better trade-off between precision and recall, whereas the best pipeline with the FLAKEFLAGGER feature set had significantly greater precision than recall, which was relatively poor. This result suggests that the FLAKEFLAGGER pipeline was particularly conservative with regard to labelling a test case as flaky. The bottom half of Table IV gives the best pipelines for the OD classification problem. In this case, the best pipeline with FLAKE16 had an F1 score that was 17% greater than the best pipeline with the FLAKEFLAGGER feature set. Table VI gives the per-project scores for these two pipelines. Once again, we excluded projects if we could not calculate precision, recall, or F1 score. For 11 of the 18 projects listed, the F1 score was greater when using FLAKE16. Overall, the best pipeline with FLAKE16 had a recall that was 36% greater than that of the FLAKEFLAGGER feature set and a precision that was unchanged, indicating a clear improvement in the performance of the flaky test detection method.

Conclusion for RQ1. The FLAKE16 feature set offered a 13% increase in overall F1 score when detecting NOD flaky tests and a 17% increase when detecting OD flaky tests. These results indicate that the FLAKE16 feature set improves machine learning-based flaky test detection performance compared to the FLAKEFLAGGER feature set.

TABLE VI
FOR OD FLAKY TESTS, THE PER-PROJECT COMPARISON OF THE BEST PIPELINE FOR BOTH FEATURE SETS.

In this table, FP, FN, and TP are false positives, false negatives, and true positives, respectively. Finally, Pr stands for precision and Re is recall.

	FLAKEFLAGGER						FLAKE16					
Project	**FP**	**FN**	**TP**	**Pr**	**Re**	**F1**	**FP**	**FN**	**TP**	**Pr**	**Re**	**F1**
airflow	89	94	199	0.69	0.68	0.69	133	55	238	0.64	0.81	0.72
celery	6	7	8	0.57	0.53	0.55	4	9	6	0.60	0.40	0.48
conan	23	7	6	0.21	0.46	0.29	14	8	5	0.26	0.38	0.31
Flexget	2	3	1	0.33	0.25	0.29	0	3	1	1.00	0.25	0.40
fonttools	19	5	37	0.66	0.88	0.76	21	1	41	0.66	0.98	0.79
hydra	19	9	10	0.34	0.53	0.42	4	13	6	0.60	0.32	0.41
ipython	96	248	56	0.37	0.18	0.25	274	126	178	0.39	0.59	0.47
kombu	6	13	10	0.62	0.43	0.51	1	15	8	0.89	0.35	0.50
libcloud	62	91	42	0.40	0.32	0.35	103	78	55	0.35	0.41	0.38
loguru	4	2	19	0.83	0.90	0.86	6	6	15	0.71	0.71	0.71
mitmproxy	10	11	6	0.38	0.35	0.36	5	12	5	0.50	0.29	0.37
Pillow	14	20	6	0.30	0.23	0.26	21	11	15	0.42	0.58	0.48
prefect	9	16	4	0.31	0.20	0.24	1	16	4	0.80	0.20	0.32
PyGithub	0	1	3	1.00	0.75	0.86	1	1	3	0.75	0.75	0.75
scikit0image	31	3	9	0.23	0.75	0.35	2	7	5	0.71	0.42	0.53
seaborn	11	6	2	0.15	0.25	0.19	1	6	2	0.67	0.25	0.36
setuptools	5	5	18	0.78	0.78	0.78	4	7	16	0.80	0.70	0.74
xonsh	10	12	7	0.41	0.37	0.39	8	13	6	0.43	0.32	0.36
Total	440	569	443	0.50	0.44	0.47	608	401	611	0.50	0.60	0.55

RQ2. Can machine learning models be applied to effectively detect order-dependent flaky test cases? As shown in the bottom half of Table IV, the most performant pipeline for detecting OD flaky tests used the FLAKE16 feature set with scaling for preprocessing, SMOTE for balancing, and random forest as the model, and achieved an F1 score of 0.55. Compared to the best NOD pipeline, its overall precision was 14% lower and its recall was 25% higher, resulting in an F1 score that was just 6% higher. These differences are too marginal to conclude that machine learning models are any better at detecting OD flaky tests than NOD flaky tests, but rather suggests that their performance at both classification problems was roughly the same. For the best OD pipeline, the per-project F1 scores showed a significant degree of variance, achieving an F1 score of just 0.31 for CONAN and up to 0.79 for FONTTOOLS. Comparatively, the best NOD pipeline had a much lower per-project variance, though the sample size of projects in this case is rather small to draw any reliable conclusions. This per-project variance is not unique to our study [3], [8], however ours is the first to report it in the context of using machine learning to detect OD flaky tests.

Conclusion for RQ2. The performance of the best OD pipeline was broadly similar to that of the best NOD pipeline, suggesting that machine learning models are just as applicable to the task of detecting OD flaky tests as they are to detecting NOD flaky tests.

RQ3. Which features of FLAKE16 are the most impactful? Figure 2 shows each feature of FLAKE16 in descending order of their mean absolute SHAP value in the context of detecting both NOD and OD flaky tests (see Section III-E

for details on how these are calculated). The lines connecting the boxes indicate how their ranks differ between the two classification problems. As indicated by the volume of lines with steep gradients, the difference is significant. For detecting NOD flaky tests, the maximum threads feature was the most impactful by a considerable margin. For OD flaky tests, the number of read- and write-related system calls were the most impactful metrics. All these features are exclusive to FLAKE16, which could partially explain why it improved detection performance compared to the FLAKEFLAGGER feature set. In general, the dynamic features occupy the higher ranks and the static features occupy the lower ranks for both classification problems. This shows that the static features had less influence on the models' decisions, implying that they may be less useful for detecting flaky tests. Clear exceptions to this are the AST depth feature, which was the second most impactful for NOD flaky tests, the number of assertions, which was third for OD flaky tests, and test lines of code, which occupied the lower-middle ranks in both instances.

Conclusion for RQ3. The most impactful feature when detecting NOD flaky tests was the peak number of concurrently running threads during test case execution. When detecting OD flaky tests, the number of read- and write-related system calls were the most impactful. In general, the dynamic features were more impactful than the static features, though there were notable exceptions to this.

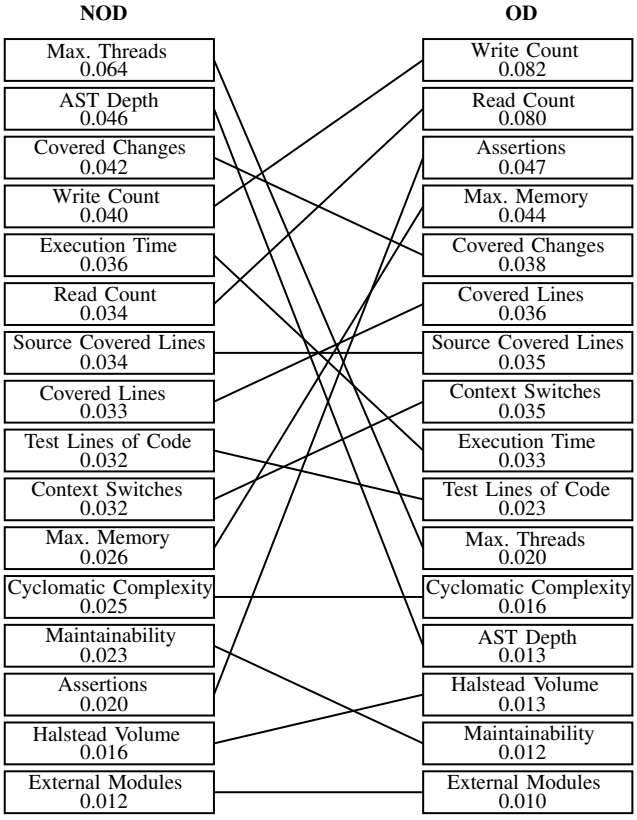

Fig. 2. Impact of each feature of FLAKE16 for both classification problems. Each box represents a feature and contains its mean absolute SHAP value (see Section III-E), of which they are in descending order. The left column contains the values in the context of detecting NOD flaky tests and the right in the context of detecting OD flaky tests. Features are connected between each column with a line, representing how significantly the ranks differ.

V. DISCUSSION

A. General Model Performance

The best pipeline for detecting NOD flaky tests achieved an F1 score of 0.52 and the best for OD flaky tests achieved an F1 score of 0.55. For a binary classification problem with balanced classes, an F1 score of 0.50 can be trivially attained by randomly guessing labels with uniform class probabilities. Yet, both of this paper's classification problems are significantly imbalanced. For the NOD problem, flaky tests account for just 0.02% of the entire dataset. For the OD problem, flaky tests represent 1.5%. In both cases, we would expect uniform random guessing to yield an F1 score significantly lower than 0.5. Considering the NOD problem, we would expect random guessing to render half of the 66,861 non-flaky test cases as true negatives and half as false positives. Similarly, we would expect half of the 145 flaky test cases to become false negatives and the other half true positives. Applying the calculations for precision and recall described in Section III-E, this strategy would score 0.5 and 0.002 respectively, giving an ultimate F1 score of 0.004. The F1 score would be similarly low for the OD problem and for both problems using other trivial approaches, such as guessing according to class prior probabilities or labelling all test cases as non-flaky. Therefore, the two pipelines that use FLAKE16 are significantly better suited to flaky test detection than these trivial approaches.

Alshammari et al. [3], who presented the FLAKEFLAGGER framework, recorded an F1 score of 0.66 when detecting NOD flaky tests. This is considerably higher than the F1 score achieved by the best pipeline with the FLAKEFLAGGER feature set in this paper's study, which was 0.46. Yet, Alshammari et al. used an entirely different dataset of tests from projects in the Java programming language, which makes the comparison largely invalid. With our Python dataset, we demonstrated that FLAKE16 improved NOD flaky test detection performance. There is no evidence to suggest that this would not also be the case on Alshammari et al.'s dataset. Since this is the first study to apply machine learning models to detecting OD flaky tests, there are no previous results to which we can compare.

B. Reliability of Performance Metrics

Extrapolating the curves of Figure 1 suggests that we would identify more flaky tests of both categories if we had FLAKE16FRAMEWORK perform more test suite runs. Since more runs can only result in test case labels transitioning from non-flaky to flaky, true negatives may become false negatives and false positives may become true positives. The effect that this would have on precision, recall, and F1 score would entirely depend on the frequency of both types of change. However, we do not consider it likely that more test suite runs would change the overall conclusion of **RQ1** for two reasons. First, any label transitions would affect the

results for the FLAKEFLAGGER feature set the same as they would for FLAKE16. Second, given the sublinear curves, we would expect to identify increasingly fewer flaky tests as FLAKE16FRAMEWORK performed more runs, meaning that any changes in the F1 scores would be increasingly small.

C. Impact of Features

Our results for **RQ3** indicate that maximum threads was the most impactful feature when detecting NOD flaky tests. This is unsurprising given the prevalence of flaky tests caused by asynchronicity and concurrency, as reported in the literature [18], [31], [38], [50]. Naturally, both of these causes imply multiple running threads during test case execution. For OD flaky tests, our results indicate that the number of read- and write-related system calls were the most impactful. Similarly, this could be explained by the relationship between filesystem activity and OD flaky tests that has been described in previous studies [6], [9], [19], [38], [66]. Interestingly, these two features were also highly impactful when detecting NOD flaky tests. A possible explanation for this is that input and output operations may be performed asynchronously [4], which has been established as a common cause of NOD flaky tests [38].

Alshammari et al.'s evaluation suggested that execution time was the most informative feature that they considered for their FLAKEFLAGGER framework [3]. In our results, execution time was the fifth most impactful feature when detecting NOD flaky tests, but was considerably less impactful for OD flaky tests. They also determined that the three coverage features were highly informative. Similarly, we found these to occupy the upper-middle ranks for both classification problems, supporting the notion that they are valuable features for detecting flaky tests. In general, we would expect features such as execution time, the three coverage features, and maximum memory to be higher for larger and more complex test cases. We hypothesize that the impactfulness of this group of features is simply a consequence of the intuition that the more a test case is doing, the greater the margin for both error and flakiness.

Many of the static features of FLAKE16, such as cyclomatic complexity, Halstead volume, and maintainability, appeared to have the lowest impact for both classification problems. A recent study cast doubt on the fitness for purpose of Halstead volume and other code complexity metrics [43]. This could be the reason why they appeared to be of limited value in the context of flaky test detection. This is despite another recent study finding that the Halstead volume of flaky tests was greater than non-flaky tests to a statically significant degree, albeit with a small effect size [45]. With this result in mind, concluding that static features are generally less valuable than dynamic features could be misguided, especially since we identified three static features that appeared to make a considerable contribution to the models' predictions.

D. Impact of Preprocessing, Balancing, and Model Choice

Table IV does not indicate any clear best choice of preprocessing or balancing. For both the FLAKEFLAGGER and FLAKE16 feature sets, pipelines with different preprocessing and balancing are in the top 10 for both classification problems with only small differences in F1 score. Most interestingly, given the significant class imbalances, pipelines with no data balancing are in the top 10 for both classification problems with the FLAKEFLAGGER features. In general, these results indicate that extra trees was the better choice of model, though this is not definitive since random forest was best for detecting OD flaky tests with FLAKE16, though in comparison to extra trees the difference in F1 score is minimal. Overall, the only reliable conclusions we can draw from these findings is that data balancing mostly improves detection performance, though there is no clear best technique, and extra trees appears to have a slight edge over random forest. When compared to random forest, extra trees trades increased *bias* for reduced *variance* [20]. Having increased bias means the model may fail to recognize relationships between feature data and labels, known as *underfitting*. Having reduced variance means the model may be less sensitive to noise and outliers, avoiding *overfitting*. It could be that the particular bias-variance tradeoff of extra trees makes it generally more suited to the specific problem of using machine learning for flaky test detection.

E. Implications

Researchers. This paper shows that using FLAKE16 improved machine learning-based flaky test detection performance compared to the FLAKEFLAGGER dataset. This demonstrates that measuring a greater diversity of test case properties allows models to better distinguish between flaky and non-flaky test cases. The results also reveal that machine learning models are just as applicable to detecting OD flaky tests as they are to detecting NOD flaky tests. Therefore, researchers should consider how machine learning models can improve the scalability of OD flaky test detection, since many previous techniques incur a significant time cost [19], [32], [66].

Developers. This paper establishes that the maximum number of concurrently running threads during test case execution is a very impactful feature when detecting NOD flaky tests. As such, our advice to developers would be to avoid concurrency in tests as much as is possible. When developers cannot heed this advice, it may be useful for them to assume such tests are likely to be flaky [25]. The same can be said for test cases that perform significant input and output, given that we found the number of read- and write- related system calls to be impactful features for detecting both NOD and OD flaky tests.

VI. RELATED WORK

One of the earliest empirical studies of flaky tests was performed by Luo et al. [38]. They classified the flaky tests repaired in 201 commits into 10 cause categories. The most common cause they identified was related to waiting for asynchronous calls. For example, a test case that launches a separate process to do some work and waits for a fixed amount of time for it to finish may fail when the process happens to take longer than expected. Another common category was *test-order dependency*, the cause of OD flaky tests. Luo et al. found that this was often characterized by the OD flaky test expecting

a particular value of a global variable which is modified by another test case, which came to be known in later studies as a *polluter* [55]. They also identified OD flaky tests that were caused by a polluter modifying a file. Subsequent studies generally support the finding that these three particular causes are very prevalent in many projects [18], [31], [38], [50].

Lam et al. [32] presented iDFlakies, a technique for detecting flaky tests and labelling them as OD or NOD. Initially, iDFlakies repeatedly executes a test suite in its *original* test run order, the default order scheduled by the test runner, to identify which test cases pass consistently. It then repeatedly executes the test suite in *modified* orders. When a test case that had consistently passed in the original order fails in a modified order, iDFlakies executes the test suite again in both the original and the modified order, up to and including the failing test case. Should the test case fail again in the modified order, but pass in the original order, the tool labels it as OD, otherwise it is labelled as NOD.

Since iDFlakies requires many repeated test executions, it may not scale well to either large or slow-running test suites. Bell et al. [7] presented DeFlaker, which, unlike iDFlakies, cannot identify OD flaky tests. Should a test case fail, having passed on a previous version of the software under test and without covering any modified code, DeFlaker labels it as flaky. To measure coverage, DeFlaker requires instrumentation. Likewise, Flake16Framework requires instrumentation to measure coverage and other metrics in Flake16. In both cases, this instrumentation introduces runtime overhead. However, Flake16Framework only requires a single instrumented run to detect flaky tests, whereas DeFlaker requires instrumentation every time it is used.

One of the earliest techniques for specifically detecting OD flaky tests was DTDetector, presented by Zhang et al. [66]. Their approach utilizes repeated test suite executions in different orders and, in some configurations, with test case isolation using separate processes. One configuration of their technique uses byte-code instrumentation to filter test cases that are unlikely to be OD flaky tests. The combination of all these factors meant the time cost of the technique was significant. Bell et al. [6] presented ElectricTest, which, unlike DTDetector, involved only a single instrumented test suite run and was thus significantly faster. The instrumentation employed by ElectricTest identifies instances where one test case would modify a location in memory that was accessed by another test case. However, there is no guarantee that this would result in a OD flaky test, and so while Bell et al. were able to demonstrate that the recall of ElectricTest was at least as good as DTDetector, its precision may be much poorer. To that end, Gambi et al. [19] presented PraDet, which uses similar instrumentation to ElectricTest. Unlike ElectricTest, PraDet verifies suspected OD flaky tests by executing subsets of the test suite in particular orders. Naturally, this made it a lot slower than ElectricTest.

The drawbacks of previous techniques, specifically the high volume of test executions, motivated several studies to evaluate machine learning models for detecting flaky tests. Bertolino et al. [8] presented FLAST for predicting if a test case is flaky based purely on its source code. Their technique uses a *k-nearest neighbor* classifier [29] which labels test cases based on their cosine distance to labelled training instances within a *bag-of-words* feature space. The bag-of-words approach is used to represent test cases as sparse vectors where each element corresponds to the frequency of a particular identifier or keyword in its source code. Pinto et al. [44] performed a similar study but included additional static features beyond the bag-of-words representation such as the number of lines of code that make up a test case. This work was subsequently replicated and expanded by Haben et al. [24]. Alshammari et al. [3] presented FlakeFlagger, a random forest model encoding test cases with a feature set containing a mixture of static and dynamic features that are mostly a subset of Flake16. Their evaluation showed that their feature set offered a 347% improvement in overall F1 score compared to Pinto et al.'s purely static feature set at the relatively minimal cost of the single test suite run, required to collect the dynamic features. One aspect these three studies have in common is that they used datasets based on Bell et al.'s evaluation of DeFlaker [7]. Recall that DeFlaker does not detect OD flaky tests, meaning they would be labelled as *non-flaky* during the training and evaluation of the models in these studies.

VII. Conclusions and Future Work

This paper presented Flake16, a new feature set that encodes test cases for machine-learning-based flaky tests detection. It evaluated the performance of 54 machine learning pipelines when detecting both NOD and OD flaky tests using both Flake16 and a previously established feature set. For both categories of flaky test, experiments with flaky tests from 26 real-world Python projects showed greater detection performance when using Flake16. Offering a more complete evaluation of the problem of flaky test detection, this paper is the first to apply machine learning models to the detection of OD flaky tests. Using the SHAP technique to evaluate the impact that each Flake16 feature has on the models' decisions, the results show that the peak number of concurrently running threads during test case execution to be the most impactful for detecting NOD flaky tests. For OD flaky tests, the number of read- and write-related system calls have the greatest impact. The experiments also reveal that static code complexity features such as cyclomatic complexity, Halstead volume, and maintainability to have little impact in both cases.

As future work, we plan to repeat our experiments with a much larger dataset of flaky tests, thereby improving the generalizability of this paper's findings. We will also include test cases from projects implemented in different programming languages. Furthermore, we will evaluate the performance of machine learning models for detecting flaky tests in additional specific categories beyond OD flaky tests. By extending our work in these three areas, we aim to create an automated technique for detecting a greater variety of flaky test types that would be applicable to many different types of project.

REFERENCES

[1] H. Abdi and L. J Williams. Principal component analysis. *Wiley Interdisciplinary Reviews: Computational Statistics*, 2(4):433–459, 2010.

[2] R. Al-Qutaish and A. Abran. *Halstead Metrics: Analysis of their Design*, pages 145–159. Wiley, 2010.

[3] A. Alshammari, C. Morris, M. Hilton, and J. Bell. FlakeFlagger: Predicting flakiness without rerunning tests. In *Proceedings of the International Conference on Software Engineering (ICSE)*, 2021.

[4] Asynchronous I/O https://docs.python.org/3/library/asyncio.html, 2022.

[5] G. E. A. P. A. Batista, R. C. Prati, and M. C. Monard. A study of the behavior of several methods for balancing machine learning training data. *Explorations Newsletter*, 6(1):20–29, 2004.

[6] J. Bell, G. Kaiser, E. Melski, and M. Dattatreya. Efficient dependency detection for safe Java test acceleration. In *Proceedings of the Joint Meeting of the European Software Engineering Conference and the Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 770–781, 2015.

[7] J. Bell, O. Legunsen, M. Hilton, L. Eloussi, T. Yung, and D. Marinov. DeFlaker: Automatically detecting flaky tests. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 433–444, 2018.

[8] A. Bertolino, E. Cruciani, B. Miranda, and R. Verdecchia. Know your neighbor: Fast static prediction of test flakiness. *IEEE Access*, 9:76119–76134, 2021.

[9] M. Biagiola, A. Stocco, A. Mesbah, F. Ricca, and P. Tonella. Web test dependency detection. In *Proceedings of the Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 154–164, 2019.

[10] L. Breiman. Random forests. *Machine Learning*, 45(1):5–32, 2001.

[11] J. Candido, L. Melo, and M. 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 (ASE)*, pages 153–158, 2017.

[12] N. V. Chawla, K. W. Bowyer, L. O. Hall, and W. P. Kegelmeyer. SMOTE: Synthetic minority over-sampling technique. *Journal of Artificial Intelligence Research*, 16:321–357, 2002.

[13] Classes, https://docs.python.org/3/tutorial/classes.html, 2022.

[14] Coverage.py documentation, https://coverage.readthedocs.io/en/stable/, 2022.

[15] Docker documentation, https://docs.docker.com/, 2022.

[16] T. Durieux, C. L. Goues, M. Hilton, and R. Abreu. Empirical study of restarted and flaky builds on Travis CI. In *Proceedings of the International Conference on Mining Software Repositories (MSR)*, pages 254–264, 2020.

[17] S. Dutta, A. Shi, R. Choudhary, Z. Zhang, A. Jain, and Misailovic S. Detecting flaky tests in probabilistic and machine learning applications. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, pages 211–224, 2020.

[18] M. Eck, F. Palomba, M. Castelluccio, and A. Bacchelli. Understanding flaky tests: The developer's perspective. In *Proceedings of the Joint Meeting of the European Software Engineering Conference and the Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 830–840, 2019.

[19] A. Gambi, J. Bell, and A. Zeller. Practical test dependency detection. In *Proceedings of the International Conference on Software Testing, Verification and Validation (ICST)*, pages 1–11, 2018.

[20] P Geurts, D. Ernst, and L. Wehenkel. Extremely randomized trees. *Machine Learning*, 63(1):3–42, 2006.

[21] G. K. Gill and C. F. Kemerer. Cyclomatic complexity density and software maintenance productivity. *Transactions on Software Engineering*, 17(12):1284, 1991.

[22] S. Grafberger, J. Stoyanovich, and S. Schelter. Lightweight inspection of data preprocessing in native machine learning pipelines. In *Proceedings of the Conference on Innovative Data Systems Research (CIDR)*, 2021.

[23] M. Gruber, S. Lukasczyk, F. Kroiß, and G. Fraser. An empirical study of flaky tests in Python. In *Proceedings of the International Conference on Software Testing, Verification and Validation (ICST)*, 2021.

[24] G. Haben, S. Habchi, M. Papadakis, M. Cordy, and Y. Le Traon. A replication study on the usability of code vocabulary in predicting flaky tests. In *Proceedings of the International Conference on Mining Software Repositories (MSR)*, 2021.

[25] M. Harman and P. O'Hearn. From start-ups to scale-ups: Opportunities and open problems for static and dynamic program analysis. In *Proceedings of the International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 1–23, 2018.

[26] G. Hooker and L. Mentch. Please stop permuting features: An explanation and alternatives. *ArXiv*, 2019.

[27] Imbalanced-Learn documenation, https://imbalanced-learn.org/stable/index.html, 2022.

[28] I/O statistics fields, https://www.kernel.org/doc/Documentation/iostats.txt, 2022.

[29] J. M. Keller, M. R. Gray, and J. A. Givens. A fuzzy k-nearest neighbor algorithm. *Transactions on Systems, Man, and Cybernetics*, 15(4):580–585, 1985.

[30] 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 International Symposium on Software Testing and Analysis (ISSTA)*, pages 204–215, 2019.

[31] W. Lam, K. Muşlu, H. Sajnani, and S. Thummalapenta. A study on the lifecycle of flaky tests. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 1471–1482, 2020.

[32] W. Lam, R. Oei, A. Shi, D. Marinov, and T. Xie. IDFlakies: A framework for detecting and partially classifying flaky tests. In *Proceedings of the International Conference on Software Testing, Verification and Validation (ICST)*, pages 312–322, 2019.

[33] W. Lam, A. Shi, R. Oei, S. Zhang, M. D. Ernst, and T. Xie. Dependent-test-aware regression testing techniques. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, pages 298–311, 2020.

[34] 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 *Proceedings of the International Conference on Software Reliability Engineering (ISSRE)*, pages 403–413, 2020.

[35] J. Listfield. Where do our flaky tests come from?, https://testing.googleblog.com/2017/04/where-do-our-flaky-tests-come-from.html, 2022.

[36] V. López, A. Fernández, S. García, V. Palade, and F. Herrera. An insight into classification with imbalanced data: Empirical results and current trends on using data intrinsic characteristics. *Information Sciences*, 250:113–141, 2013.

[37] S. M. Lundberg, G. Erion, H. Chen, A. DeGrave, J. M. Prutkin, B. Nair, R. Katz, J. Himmelfarb, N. Bansal, and S. Lee. From local explanations to global understanding with explainable AI for trees. *Nature Machine Intelligence*, 2(1):2522–5839, 2020.

[38] Q. Luo, F. Hariri, L. Eloussi, and D. Marinov. An empirical analysis of flaky tests. In *Proceedings of the Symposium on the Foundations of Software Engineering (FSE)*, pages 643–653, 2014.

[39] M. Machalica, A. Samylkin, M. Porth, and S. Chandra. Predictive test selection. In *Proceedings of the International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pages 91–100, 2019.

[40] A. Memon, Z. Gao, B. Nguyen, S. Dhanda, E. Nickell, R. Siemborski, and J. Micco. Taming Google-scale continuous testing. In *Proceedings of the International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pages 233–242, 2017.

[41] Open source project criticality score, https://github.com/ossf/criticality_score, 2022.

[42] O. Parry, G. M. Kapfhammer, M. Hilton, and P. McMinn. Surveying the developer experience of flaky tests. In *Proceedings of the International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, 2022.

[43] N. Peitek, S. Apel, C. Parnin, A. Brechmann, and J. Siegmund. Program comprehension and code complexity metrics: An fMRI study. In *International Conference on Software Engineering (ICSE)*, pages 524–536, 2021.

[44] G. Pinto, B. Miranda, S. Dissanayake, M. D. Amorim, C. Treude, A. Bertolino, and M. D'amorim. What is the vocabulary of flaky tests? In *Proceedings of the International Conference on Mining Software Repositories (MSR)*, pages 492–502, 2020.

[45] V. Pontillo, F. Palomba, and F. Ferrucci. Toward static test flakiness prediction: A feasibility study. In *Proceedings of the International Workshop on Machine Learning Techniques for Software Quality Evoluton*, pages 19–24, 2021.

[46] Preprocessing data https://scikit-learn.org/stable/modules/preprocessing.html#preprocessing-scaler, 2022.

[47] Psutil documentation, https://psutil.readthedocs.io/en/stable/, 2022.

[48] Radon documentation, https://radon.readthedocs.io/en/stable/index.html, 2022.

[49] Replication package, https://github.com/flake-it/flake16-framework, 2022.

[50] A. Romano, Z. Song, S. Grandhi, W. Yang, and W. Wang. An empirical analysis of UI-based flaky tests. In *Proceedings of the International Conference on Software Engineering (ICSE)*, 2021.

[51] S. R. Safavian and D. Landgrebe. A survey of decision tree classifier methodology. *Transactions on Systems, Man, and Cybernetics*, 21(3):660–674, 1991.

[52] Saving repositories with stars https://docs.github.com/en/get-started/exploring-projects-on-github/saving-repositories-with-stars, 2022.

[53] Scikit-Learn documentation, https://scikit-learn.org/stable/, 2022.

[54] A. Shi, A. Gyori, O. Legunsen, and D. Marinov. Detecting assumptions on deterministic implementations of non-deterministic specifications. In *Proceedings of the International Conference on Software Testing, Verification and Validation (ICST)*, pages 80–90, 2016.

[55] A. Shi, W. Lam, R. Oei, T. Xie, and D. Marinov. iFixFlakies: A framework for automatically fixing order-dependent flaky tests. In *Proceedings of the Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 545–555, 2019.

[56] T. Shi and S. Horvath. Unsupervised learning with random forest predictors. *Journal of Computational and Graphical Statistics*, 15(1):118–138, 2006.

[57] D. Silva, L. Teixeira, and M. D'Amorim. Shake it! Detecting flaky tests caused by concurrency with Shaker. In *Proceedings of the International Conference on Software Maintenance and Evolution (ICSME)*, pages 301–311, 2020.

[58] Y. Sun, A. K. C. Wong, and M. S. Kamel. Classification of imbalanced data: A review. *International Journal of Pattern Recognition and Artificial Intelligence*, 23(4):687–719, 2009.

[59] Ivan Tomek. Two modifications of cnn. *Transactions on Systems, Man, and Cybernetics*, 6:769–772, 1976.

[60] M. Tufano, F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, A. De Lucia, and D. Poshyvanyk. An empirical investigation into the nature of test smells. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*, pages 4–15, 2016.

[61] A. Vahabzadeh, A. A. Fard, and A. Mesbah. An empirical study of bugs in test code. In *Proceedings of the International Conference on Software Maintenance and Evolution (ICSME)*, pages 101–110, 2015.

[62] L. Van Der Maaten, E. Postma, and J. Van den Herik. Dimensionality reduction: A comparative. *Journal of Machine Learning Research*, 10(66-71):13, 2009.

[63] Virtual environments and packages, https://docs.python.org/3/tutorial/venv.html, 2022.

[64] K. D Welker. The software maintainability index revisited. *CrossTalk*, 14:18–21, 2001.

[65] C. V. G. Zelaya. Towards explaining the effects of data preprocessing on machine learning. In *Proceedings of the International Conference on Data Engineering (ICDE)*, pages 2086–2090, 2019.

[66] 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 International Symposium on Software Testing and Analysis (ISSTA)*, pages 385–396, 2014.