Noname manuscript No.
(will be inserted by the editor)

Empirically Evaluating Flaky Test Detection Techniques Combining Test Case Rerunning and Machine Learning Models

Owain Parry · Gregory M. Kapfhammer · Michael Hilton · Phil McMinn

Received: date / Accepted: date

Abstract A *flaky test* is a test case whose outcome changes without modification to the code of the test case or the program under test. These tests disrupt continuous integration, cause a loss of developer productivity, and limit the efficiency of testing. Many flaky test detection techniques are *rerunning-based*, meaning they require repeated test case executions at a considerable time cost, or are *machine learning-based*, and thus they are fast but offer only an approximate solution with variable detection performance. These two extremes leave developers with a stark choice. This paper introduces CANNIER, an approach for reducing the time cost of rerunning-based detection techniques by combining them with machine learning models. The empirical evaluation involving 89,668 test cases from 30 Python projects demonstrates that CANNIER can reduce the time cost of existing rerunning-based techniques by an order of magnitude while maintaining a detection performance that is significantly better than machine learning models alone. Furthermore, the comprehensive study extends existing work on machine learning-based detection and reveals a number of additional findings, including (1) the performance of machine learning models for detecting polluter test cases; (2) using the mean values of dynamic test case features from repeated measurements can slightly improve the detection performance of machine learning models; and (3) correlations between various test case features and the probability of the test case being flaky.

Keywords Software Testing · Flaky Tests · Machine Learning

Owain Parry
University of Sheffield, UK E-mail: oparry1@sheffield.ac.uk

Gregory M. Kapfhammer
Allegheny College, USA E-mail: gkapfham@allegheny.edu

Michael Hilton
Carnegie Mellon University, USA E-mail: mhilton@cmu.edu

Phil McMinn
University of Sheffield, UK E-mail: p.mcminn@sheffield.ac.uk

1 Introduction

A *flaky test* is a test case that can exhibit both passing and failing behavior without changes to the code of the test case or the program under test [54]. They are a serious problem for software developers because they disrupt continuous integration, cause a loss of productivity, and limit the efficiency of testing. The pain of flaky tests is felt by developers in both the open-source domain [34] and in large companies such as Google, Microsoft, and Facebook [45,51,52]. A survey of developers found that 56% observed flaky tests on at least a monthly basis in the projects on which they were currently working [56].

Flaky tests that depend on the prior execution of other test cases in the test run order are known as *order-dependent* flaky tests. Another term for such flaky tests is *victim*, and the prior test cases that affect their outcome are known as *polluters* [65][1]. Victim flaky tests are very prevalent, with one study finding that 51% of the 422 flaky tests in 82 Java projects were victims [47]. They are a major snag to techniques that split-up or reorder a test suite, such as test case prioritization, selection, and parallelization [23,30,48].

The research community has introduced a multitude of automated techniques to detect flaky tests. Many are *rerunning-based*, meaning they may require an excessive number of repeated test case executions, making them expensive for deployment in large software projects [47,76]. Alshammari et al. [22] repeatedly executed the test suites of 24 Java projects and were still detecting *non-order-dependent* (NOD) flaky tests after 10,000 reruns. We estimated that the single-core time cost of detecting the 158 NOD flaky tests in our subject set of 89,668 test cases by rerunning them up to 2,500 times is 1.6 years. The time cost of rerunning-based detection led researchers to investigate techniques that do not require test case runs but are instead based on machine learning models trained on features of the test case code [25,58]. Later studies found that combining these with dynamic test case features, such as execution time and line coverage, increases detection performance at the cost of a single instrumented test suite run to measure these features [22,55]. Despite this, machine learning models in this domain offer only an approximate solution. For example, for detecting NOD flaky tests in Java projects, one previous study's evaluation shows a Matthews correlation coefficient (MCC), a reliable metric for evaluating a machine learning model [32], of 0.65 [22]. Another, focusing on Python projects, shows an MCC of 0.53 [55]. For these results, we would expect a perfect machine learning model to score 1 and a model no better than random guessing to score 0. The prohibitive time cost of rerunning-based techniques and the limited performance of machine learning-based techniques leaves practitioners with a stark choice when it comes to detecting flaky tests.

This paper introduces CANNIER (maChine leArNiNg assIsted tEst Rerunning), a high-level approach for reducing the time cost of rerunning-based detection techniques by combining them with machine learning models.

[1] The paper that introduced the victim/polluter terminology [65] also introduced the terms *brittle*/*state-setter* for when the order-dependent test fails in isolation. In this paper, we use the victim/polluter terms generally, regardless of the order-dependent test's outcome.

It does this by using the output of the models as a heuristic to reduce the problem space for the rerunning-based technique. We demonstrate the applicability of CANNIER by instantiating it for three previously established detection techniques. We implemented these within an automated tool and empirically evaluated them using 30 Python projects as subjects. We found that CANNIER could significantly reduce time cost at the expense of only a minor reduction in detection performance. For example, by applying CANNIER to the Classification stage of iDFlakies [47] (that distinguishes NOD flaky tests from victim flaky tests), we were able to reduce its time cost by 84% at the expense of misclassifying just 8 flaky tests out of 1,130. Therefore, CANNIER represents a "best of both worlds" solution to flaky test detection. In summary, the main contributions of this paper are:

Contribution 1: Approach. A novel approach, called CANNIER, that significantly reduces the time cost of rerunning-based flaky test detection with a minimal decrease in detection performance. See Section 3 for more details.

Contribution 2: Tooling. To facilitate our empirical evaluation and allow for replication of our results, we developed an extensive framework of automated tools that we make freely available [5]. See Section 4 for more details.

Contribution 3: Empirical Evaluation. A comprehensive empirical evaluation demonstrates the effectiveness of CANNIER's combination of re-running and machine learning techniques, revealing further novel findings about machine learning-based flaky test detection, such as the performance of machine learning models for detecting polluter test cases. See Section 5 for more details.

Contribution 4: Dataset. A dataset containing 89,668 tests from 30 Python projects taking over six weeks of compute time to produce. We make this available as part of our replication package [4]. See Section 5.1 for more details.

2 Background

2.1 Rerunning-Based Flaky Test Detection

2.1.1 Rerun

The research community has presented many automated flaky test detection techniques that are based on rerunning test cases. The most straight-forward such technique is to repeatedly execute a test case until it exhibits both passing and failing behavior. In its most basic form, this technique involves rerunning the test cases of a test suite in the same test run order and under the same environmental conditions each time [24]. We refer to this specific technique as Rerun. Since the test run order remains constant, Rerun can only identify non-order-dependent (NOD) flaky tests. As its only parameter, Rerun requires an upper-limit on the number of times to execute a test case without observing an inconsistent outcome. If the upper-limit is reached, the technique classifies the test case as non-flaky and stops rerunning it. Since many test cases may require hundreds or even thousands of runs to manifest their flakiness [22], this technique can become very expensive for long-running test suites with numerous tests, thus limiting the technique in practice.

2.1.2 iDFlakies

Lam et al. [47] presented iDFlakies, a technique for detecting flaky tests and classifying them as NOD or a victim. The technique consists of three stages: *Setup*, *Running*, and *Classification*. In the Setup stage, iDFlakies repeatedly executes the test suite in its original order to identify and filter any consistently-failing test cases. In the Running stage, iDFlakies continues to rerun the test suite, but this time in modified test run orders. In the Classification stage, for every test case that failed during the Running stage, iDFlakies re-executes the test suite in both the original order and in the modified order that witnessed the failure, truncated up to and including the failing test case. We refer to this stage as iDFClass (iDF~~lakies~~Class~~ification~~). Should the test case fail again in the truncated modified order and pass again in the truncated original order, iDFlakies classifies it as a victim. Otherwise, it classifies the test case as NOD. Should a test case fail multiple times during the Running stage, iDFlakies can repeat the Classification stage for a percentage of the additional failures for greater confidence in the final label.

iDFlakies has several parameters: the number of reruns during the Setup stage, the number of reruns during the Running stage, the method of generating the modified test run orders during the Running stage (e.g., shuffle), and the percentage of additional failures to recheck in the Classification stage. Depending on the choice of values for these parameters, iDFlakies can require a significant number of test executions and thus impose a prohibitive time cost.

2.1.3 Pairwise

While the iDFlakies technique can detect victim flaky tests, it cannot identify their associated polluters. Zhang et al. [76] proposed a technique that can detect a subset of a test suite's victims and their polluters (although the authors designed the technique primarily for detecting victims). It involves executing every permutation of test cases of length two (every pair in both orders) in isolation, such as in separate Java Virtual Machine or Python interpreter processes. We refer to this technique as Pairwise. Initially, Pairwise requires an expected outcome for every test case. It could obtain these by executing each test case in isolation to observe their outcome independent of the possible side-effects of other test cases. Once every test case has an expected outcome, Pairwise executes every 2-permutation of test cases, such that each test case has a turn at being both the first and second to be executed in the pair — the candidate polluter and victim, respectively. For more reliable results, Pairwise ought to filter out any pairs with a known NOD flaky test as the candidate victim because they do not have a reliable expected outcome (although Zhang et al. did not propose this filtering stage in their paper). For a given pair, if the second test yields an outcome different from expected, PAIRWISE classifies it as a victim and classifies the first test as one of its polluters.

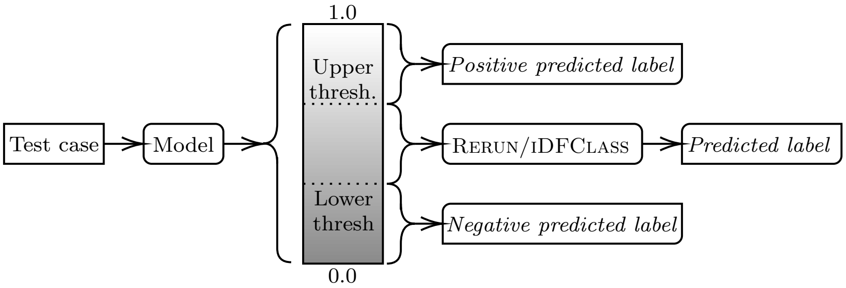
Previous work has determined that an order-dependency can involve more than two test cases [65], though as part of their empirical study, Zhang et al. found that 76% of order-dependencies did involve just two. Considering only pairs of test cases, the time complexity of PAIRWISE is already quadratic in the size of the test suite and hence very expensive, and so to consider longer permutations would quickly render the technique intractable.

2.2 The FLAKE16 Feature Set

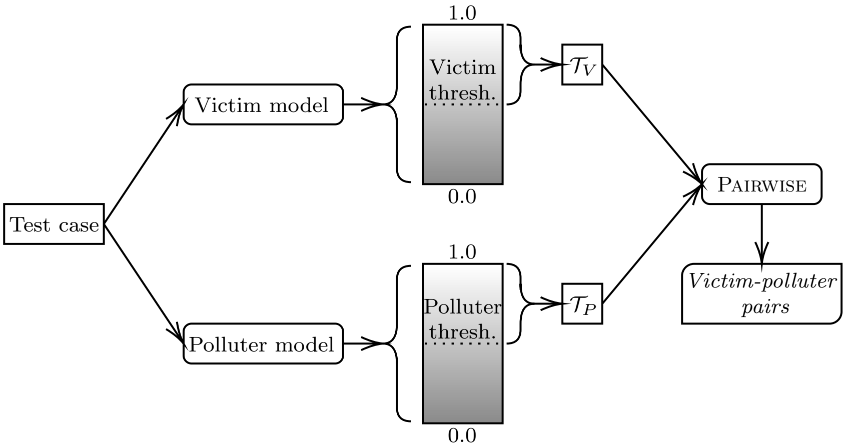
Alshammari et al. [22] introduced FLAKEFLAGGER, a tool for detecting flaky tests using a machine learning model. To encode test cases for model training and evaluation, they used a feature set initially consisting of eight numerical test case features and eight boolean features indicating the presence of test smells [37]. However, having found the eight test smell features to offer very little information gain, they eventually discarded them. Their empirical evaluation involving 24 Java projects showed that their machine learning model achieved a Matthews correlation coefficient (MCC) of 0.65. In our previous work, we introduced the FLAKE16 feature set for encoding test cases [55]. It subsumes the feature set used by the FLAKEFLAGGER tool and introduces additional metrics such as the number of times the filesystem performed input/output operations during test case execution and the peak memory usage. Our previous evaluation involving 26 Python projects showed that models based on FLAKE16 generally outperformed models based on the FLAKEFLAGGER feature set for detecting both NOD and Victim flaky tests.

3 The CANNIER Approach

CANNIER (maChine leArNiNg assIsted tEst Rerunning) is a high-level approach that combines a rerunning-based flaky test detection technique and one or more machine learning models. The models must provide a predicted probability that a given test case is flaky. The general concept behind CANNIER is to use the predicted probabilities as a heuristic to reduce the problem space for the rerunning-based technique. As attested by our later empirical evaluation (see Section 5), this approach can dramatically reduce the number of test case executions, and therefore time cost, at the expense of only a minor decrease in detection performance. The specifics of how CANNIER uses the predicted probabilities depends on the nature of the rerunning-based technique. Fig. 1 provides a visual summary of the application of CANNIER to the three rerunning-based detection techniques introduced in Section 2.1.



(a) For RERUN and IDFCLASS, CANNIER uses a single machine learning model to predict the probability of a given test case being of the positive class (flaky). If the probability is below a lower-threshold or above an upper-threshold, CANNIER assigns the test case a negative or positive predicted label respectively. In the ambiguous region between the two thresholds, CANNIER delegates to the rerunning-based technique to predict the label.



(b) For PAIRWISE, CANNIER uses one machine learning model to predict the probability of each test case being a victim and another to do the same for being a polluter. All those with a probability of being a victim above a threshold join the set of candidate victims, $\mathcal{T}_V$. Similarly, those with a probability of being a polluter above a threshold enter the candidate polluters set, $\mathcal{T}_P$. CANNIER then restricts PAIRWISE to consider only test cases from $\mathcal{T}_P$ as the first test case of each pair, and only test cases from $\mathcal{T}_V$ as the second.

Fig. 1: CANNIER uses the predicted probabilities from one or more machine learning models as a heuristic to reduce the problem space of a rerunning-based flaky test detection technique. A single machine learning model is suitable for RERUN and the Classification stage of IDFLAKIES (IDFCLASS) (a). Two machine learning models are required for PAIRWISE (b).

3.1 Motivating Example

We used the AIRFLOW project, developed by the Apache Software Foundation, as one of the subjects in our empirical evaluation [3]. Its test suite contains 3,251 test cases as of version 1.10.14. We executed the test suite 2,500 times in its original order and identified 66 NOD flaky tests. Following our empirical evaluation, we found that the single-core time cost to detect these flaky tests, using Rerun with a maximum of 2,500 reruns per test case, is 1.69×10^6 seconds. This is based on the time cost of each individual test case that we measured on a machine with a 24-core AMD Ryzen 5900X CPU. Having the same number of virtual cores and a comparable single-core performance, m5zn.6xlarge is arguably the most similar cloud instance offered by Amazon Web Services [11]. As of August 2022, Amazon offers this instance at the on-demand hourly rate of 1.982 USD. This means that to detect the NOD flaky tests in Airflow using Rerun would take $((1.69 \times 10^6) \div 24) \div 60^2 \approx 19.56$ hours and cost $19.56 \times 1.982 \approx 38.77$ USD on this instance.

Given the cost in both time and money of using Rerun to detect flaky tests, a developer may instead opt to use a machine learning model. We trained an *extra trees* model, a variation of random forest [38], to detect NOD flaky tests and evaluated it using *stratified 10-fold cross validation.* Within Airflow, we found that it misclassified 26 test cases that were flaky as non-flaky, and 26 test cases that were non-flaky as flaky. With only 40 of the 66 NOD flaky tests cases actually classified as such, the model achieved a precision of $40 \div (40 + 26) \approx 61\%$ and a recall of $40 \div (40 + 26) \approx 61\%$. In this context, precision is the percentage of detected flaky tests that are genuinely flaky and recall is the percentage of genuinely flaky tests that were detected. Therefore, machine learning-based detection offers a very approximate solution. Because the model uses dynamic features, the time cost of applying it is approximately equal to the time cost of a single test suite run to produce a feature vector for each test case. We observed a single-core time cost for this of 7.77×10^2 seconds for Airflow. This would require $((7.77 \times 10^2) \div 24) \div 60^2 \approx 0.01$ hours on an m5zn.6xlarge instance, costing $0.01 \times 1.982 \approx 0.02$ USD. We do not consider the time cost associated with applying the extra trees model to each test case. This is because it is negligible relative to the time taken to execute the test suite (typically less than one second), as we found in our previous work [55]. We also do not consider the time taken to train the extra trees model. This is because the model only needs to be trained once and can then be applied any number of times. For this reason, we consider training to be an off-line stage that does not contribute to the time cost of applying the model to test cases.

Rerunning-based detection and machine learning-based detection represent opposite extremes. As shown in this example, Rerun is very expensive and the extra trees model is cheap but very approximate. By applying CANNIER to Rerun (CANNIER+Rerun), developers get a flaky test detection technique that is much cheaper than Rerun and much more accurate than the extra trees model. Following our empirical evaluation, we found that the single-core time cost to detect the 66 NOD flaky tests in Airflow using CANNIER+Rerun is 7.71×10^5 seconds. This would require $((7.71 \times 10^5) \div 24) \div 60^2 \approx 8.92$ hours on an m5zn.6xlarge instance at a cost of $8.92 \times 1.982 \approx 17.68$ USD. Therefore, CANNIER reduces the cost in USD of Rerun by 54%. We also found that it misclassified three flaky tests as non-flaky but correctly classified the remaining 62. This leads to a precision of $63 \div (63 + 0) = 100\%$ and a recall of $63 \div (63 + 3) \approx 95\%$. This is far more accurate than the extra trees model that only achieved a precision and recall of 61%.

Our empirical evaluation demonstrates that CANNIER is effective for multiple projects and the three rerunning-based detection techniques introduced in Section 2.1. For our whole dataset of 89,668 test cases from 30 projects, we found that CANNIER was able to reduce the time cost (and therefore monetary cost) by an average of 88% across the three techniques.

3.2 Single-Model CANNIER

Using CANNIER with a single machine learning model is suitable for reducing the time cost of Rerun and iDFClass (the classification stage of iDFlakies). In the case of Rerun, the flaky test classification problem is that of distinguishing NOD flaky tests from the rest of the test cases. Since it is a binary problem, NOD flaky tests are the positive class and the rest of the test cases are the negative. For iDFClass, it is telling apart NOD and victim flaky tests. In this case, NOD flaky tests are the positive class and victims are the negative. For both Rerun and iDFClass, the machine learning model should provide a predicted probability of belonging to the positive class for each test case. CANNIER assigns a positive predicted label to a test case if this probability is above an upper threshold and a negative predicted label if it is below a lower threshold. This leaves an ambiguous region between the two thresholds. CANNIER delegates any test cases with predicted probabilities within this ambiguous region to the rerunning-based technique.

3.3 Multi-Model CANNIER

Using two models, CANNIER can reduce the time cost of Pairwise. The first model is used to predict the probability of each test case being a victim. In other words, it addresses the classification problem of distinguishing victims from non-victims. The second is used to do the same but for being a polluter. For both models, CANNIER classifies every test case above a threshold as the positive class (a victim or a polluter) and every other test case as the negative (not a victim or not a polluter). In this way, CANNIER produces two non-mutually exclusive sets, one of victims, $\mathcal{T}_V$, and one of polluters, $\mathcal{T}_P$ (there is no reason why a test case cannot be both a victim and a polluter [70]). Then, CANNIER applies Pairwise with only the members of $\mathcal{T}_P$ as the first test in each pair and only the members of $\mathcal{T}_V$ as the second. Therefore, CANNIER can reduce the time complexity of Pairwise from $O(|\mathcal{T}|^2)$, where $\mathcal{T}$ is the set of all test cases in the test suite, to $O(|\mathcal{T}_V| \times |\mathcal{T}_P|)$, that is considerably faster even when $\mathcal{T}_V$ and $\mathcal{T}_P$ are not significantly smaller than $\mathcal{T}$.

Table 1: The 18 features measured by pytest-CANNIER.

#	**Feature**	**Description**
1	Read Count	Number of times the filesystem had to perform input [9].
2	Write Count	Number of times the filesystem had to perform output [9].
3	Run Time	Elapsed wall-clock time of the whole test case execution.
4	Wait Time	Elapsed wall-clock time spent waiting for input/output operations to complete.
5	Context Switches	Number of voluntary context switches.
6	Covered Lines	Number of lines covered.
7	Source Covered Lines	Number of lines covered that are not part of test cases.
8	Covered Changes	Total number of times each covered line has been modified in the last 75 commits.
9	Max. Threads	Peak number of concurrently running threads.
10	Max. Children	Peak number of concurrently running child processes.
11	Max. Memory	Peak memory usage.
12	AST Depth	Maximum depth of nested program statements in the test case code.
13	Assertions	Number of assertion statements in the test case code.
14	External Modules	Number of non-standard modules (i.e., libraries) used by the test case.
15	Halstead Volume	A measure of the size of an algorithm's implementation [21, 57,59].
16	Cyclomatic Complexity	Number of branches in the test case code [39,57,59].
17	Test Lines of Code	Number of lines in the test case code [57,59].
18	Maintainability	A measure of how easy the test case code is to support and modify [19,71].

4 Tooling

To produce our dataset and facilitate our empirical evaluation, we developed our own suite of automated tools including a plugin for the Python testing framework pytest [15], named pytest-CANNIER [14], and a command-line tool named CANNIER-Framework [5]. The purpose of pytest-CANNIER is to add the functionality to pytest necessary for our evaluation. This includes recording test case outcomes and measuring feature values. The purpose of CANNIER-Framework is to automate every aspect of our evaluation, including executing pytest-CANNIER on the subject test suites, collating raw data, and training and evaluating machine learning models.

4.1 pytest-CANNIER

We decided to target pytest due to its compatibility with test suites written for other frameworks such as unittest [17]. pytest-CANNIER takes a test suite $\mathcal{T}$ as input and offers four execution modes: *Baseline*, *Shuffle*, *Features*, and *Victim*. In the Baseline mode, the plugin executes the test suite as normal. For each test case $t \in \mathcal{T}$, pytest-CANNIER records its outcome b_t, that is either pass, $b_t = 0$, or fail, $b_t = 1$. In the Shuffle mode, the plugin randomizes the order of the test cases and records the outcome of every test case s_t.

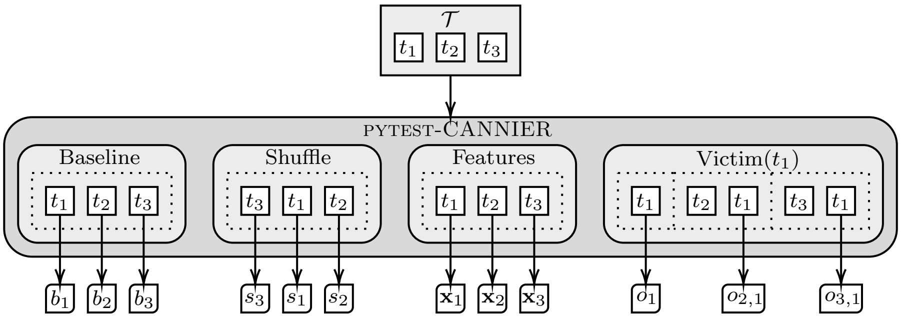


Fig. 2: As input, pytest-CANNIER takes a test suite $\mathcal{T}$=(t_1, t_2, t_3) and can be launched in four modes: *Baseline*, *Shuffle*, *Features*, or *Victim*. In the Baseline mode, the plugin runs the test suite in its original order and records the pass/fail outcome of every test case (b_1, b_2, b_3). In the Shuffle mode, pytest-CANNIER executes the test suite in a random order and also records test case outcomes (s_1, s_2, s_3). In the Features mode, the plugin produces a feature vector for each test case ($\mathbf{x}_1$, $\mathbf{x}_2$, $\mathbf{x}_3$). In the Victim mode, pytest-CANNIER takes a victim test case as an additional input (t_1) and initially executes it in isolation to ascertain its expected outcome (o_1). Then, the plugin executes every other test case in a separate process with the victim immediately following and records its outcome ($o_{2,1}$, $o_{3,1}$). This is to identify polluters of the victim.

In the Features mode, pytest-CANNIER produces a feature vector $\mathbf{x}_t \in \mathbb{R}^{18}$, for each test case t. This contains the 16 features of Flake16 alongside two additional metrics. The first of these is *Wait Time*. This is the amount of time during test case execution spent waiting for input/output (I/O) operations to complete. Previous research identified I/O in test cases as being potentially associated with flakiness [50]. The second additional feature is *Max. Children*. This measures the peak number of concurrently running child processes. A finding that many empirical studies have in common is that asynchronous operations and concurrency are very frequent causes of flaky tests [35,46,50,61]. This was our rationale for the inclusion of *Max. Threads* into Flake16. However, due to the global interpreter lock implemented within the CPython interpreter [8], it may be necessary for developers to achieve concurrency with child processes. Table 1 offers a description of all 18 features. In the Victim mode, the plugin takes a test case v, executes the test sequence $\langle v \rangle$, and records the outcome of v, o_v. This is to ascertain the expected outcome of v when executed in isolation from the rest of the test suite. Following this, pytest-CANNIER executes the sequences $\langle p, v \rangle$ for every test case p in $\mathcal{T} - \{v\}$, while recording the outcome of v when executed immediately after each p, $o_{p,v}$. This is to identify the polluters of v where $o_{p,v} \neq o_v$. For isolation between sequence runs, the plugin executes them in separate Python processes [24,76]. This implements the Pairwise technique with respect to a single candidate victim v. Fig. 2 provides a visual summary of pytest-CANNIER.

4.2 CANNIER-Framework

4.2.1 Model Training and Evaluation Data

As input, CANNIER-Framework takes a subject set of test suites $\mathcal{U}$. With every test suite $\mathcal{T} \in \mathcal{U}$ as input, the framework executes the plugin N_B times in the Baseline mode, resulting in N_B values of b_t, $(b_{t,1}, b_{t,2}, \ldots, b_{t,N_B})$, for each test case $t \in \mathcal{T}$. Similarly, CANNIER-Framework runs every test suite N_S times in the Shuffle mode, leading to N_S values of s_t, $(s_{t,1}, s_{t,2}, \ldots, s_{t,N_S})$. In both cases, the framework counts the number of times that every test case fails in the Baseline mode B_t, and the number of times in the Shuffle mode S_t. The definition of both values is given in the following equation.

$$B_t = \sum_{i=1}^{N_B} b_{t,i} \qquad S_t = \sum_{i=1}^{N_S} s_{t,i} \tag{1}$$

CANNIER-Framework also executes each test suite N_F times with pytest-CANNIER in the Features mode, resulting in N_F feature vectors for every test case t $(\mathbf{x}_{t,1}, \mathbf{x}_{t,2}, \ldots, \mathbf{x}_{t,N_F})$. As an additional input, the framework takes $\mathcal{I}$, a random sample of n_F indices ranging from 1 to N_F inclusive without replacement. With this, the framework produces a mean feature vector $\mathbf{X}_t(\mathcal{I})$, to encode each test case according to the following equation.

$$\mathbf{X}_t(\mathcal{I}) = \frac{1}{n_F} \sum_{i \in \mathcal{I}} \mathbf{x}_{t,i} \tag{2}$$

For each $\mathcal{T} \in \mathcal{U}$, the framework runs the Victim mode of pytest-CANNIER with every test case that had a consistent outcome in the Baseline mode $(B_v = 0 \vee B_v = N_B)$ and an inconsistent outcome in the Shuffle mode $(B_v \neq S_v)$ as the candidate victim v. The former condition is to ensure that every v has the reliable expected outcome that Pairwise requires. The latter is a time saving measure — if a test case is consistent in the Shuffle mode then it is very unlikely to be a victim and therefore would have no polluters. For the purposes of greater reproducibility and isolation, CANNIER-Framework executes the plugin in a separate Docker container for every run of a test suite [7]. Our Dockerfile contains all the commands needed to reproduce our Docker image and is available as part of the replication package [4].

Once the plugin has finished performing the test suite runs, CANNIER-Framework determines a *ground-truth label* $y_{t,\phi}$, for every test case t in the whole subject set, $t \in \bigcup_{\mathcal{T} \in \mathcal{U}} \mathcal{T}$, and flaky test classification problem ϕ. Recall from Section 3 that these problems are: NOD flaky tests versus the rest of the test cases (NOD-vs-Rest, $\phi = 1$), NOD flaky tests versus victim flaky tests (NOD-vs-Victim, $\phi = 2$), victim flaky tests versus the rest (Victim-vs-Rest, $\phi = 3$), and polluters versus the rest (Polluter-vs-Rest, $\phi = 4$). Each problem has a domain $\mathcal{T}_\phi \subseteq \mathcal{T}$, that is the subset of test cases in a given test suite $\mathcal{T}$ that are relevant. Since the problems are binary classifications, they also have a positive class, $\mathcal{T}_\phi^+ \subset \mathcal{T}_\phi$, and a negative class, $\mathcal{T}_\phi^- = \mathcal{T}_\phi - \mathcal{T}_\phi^+$. The ground-truth label for a test case is positive if it is in the positive class of a problem

Table 2: The four flaky test classification problems. For test suite $\mathcal{T}$, the domain $\mathcal{T}_\phi$ is the subset of $\mathcal{T}$ that is relevant to the problem ϕ. For a specific $\mathcal{T}_\phi$, the negative class for each problem ($\mathcal{T}_\phi^-$) is the complement of the positive.

ϕ **Name**	**Domain** ($\mathcal{T}_\phi$)	**Positive Class** ($\mathcal{T}_\phi^+$)
1 NOD-vs-Rest	$\mathcal{T}$	$\{t \mid t \in \mathcal{T}_\phi, 0 < B_t < N_B\}$
2 NOD-vs-Victim	$\{t \mid t \in \mathcal{T}, B_t < N_B \wedge S_t > 0\}$	$\{t \mid t \in \mathcal{T}_\phi, 0 < B_t < N_B\}$
3 Victim-vs-Rest	$\mathcal{T}$	$\{t \mid t \in \mathcal{T}_\phi, (B_t = 0 \vee B_t = N_B) \wedge B_t \neq S_t\}$
4 Polluter-vs-Rest	$\mathcal{T}$	$\{p \mid p \in \mathcal{T}_\phi, \exists v \in \mathcal{T}_\phi - \{p\}(o_{p,v} \neq o_v)\}$

($y_{t,\phi} = 1$) and negative otherwise ($y_{t,\phi} = 0$). For a test case t belonging to test suite $\mathcal{T}$, the following equation defines the ground truth label $y_{t,\phi}$.

$$y_{t,\phi} = \begin{cases} 0 & \text{if } t \in \mathcal{T}_\phi^- \\ 1 & \text{if } t \in \mathcal{T}_\phi^+ \end{cases} \tag{3}$$

For the NOD-vs-Rest problem ($\phi = 1$), the positive class is the set of NOD flaky tests, that we define as those with an inconsistent outcome in the Baseline mode ($0 < B_t < N_B$). The only test cases that are relevant to the NOD-vs-Victim problem ($\phi = 2$) are those that did not consistently fail during the runs in the Baseline mode ($B_t < N_B$) and failed at least once in Shuffle mode ($S_t > 0$). The former condition corresponds to the Setup stage of iDFlakies where such test cases would be excluded from further analysis. The latter corresponds to the Running stage, where any test case that fails at least once goes on to the Classification stage. For this problem, the positive class is also the set of NOD flaky tests. For the Victim-vs-Rest problem ($\phi = 3$), the positive class is the set of test cases with a consistent outcome in the Baseline mode ($B_t = 0 \vee B_t = N_B$) and an inconsistent outcome in the Shuffle mode ($B_t \neq S_t$). This represents the set of victims. Finally, for the Polluter-vs-Rest problem ($\phi = 4$), the positive class is the set of test cases that behaved as polluters in the Victim mode. Table 2 gives a definition of each problem.

4.2.2 Model Training and Evaluation Procedure

CANNIER-Framework follows a general machine learning pipeline for model training and evaluation. The pipeline leaves the specific model and data balancing technique unspecified, such that it can be instantiated with a choice for both of these components to create a concrete pipeline. The pipeline performs stratified 10-folds cross validation. This creates ten *folds* where 90% of the test cases in the whole subject set are for training and the other 10% are for evaluation. The class proportion of each fold roughly follows that of the whole subject set, and since that is highly imbalanced for every classification problem, the framework applies the data balancing technique to the training set only [31]. For each fold, the framework fits the machine learning model with the training set and applies it to every test case in the evaluation set. For a given problem ϕ, this results in a *predicted probability* $P(y_{t,\phi} = 1 \mid \mathbf{X}_t(\mathcal{I}))$, of

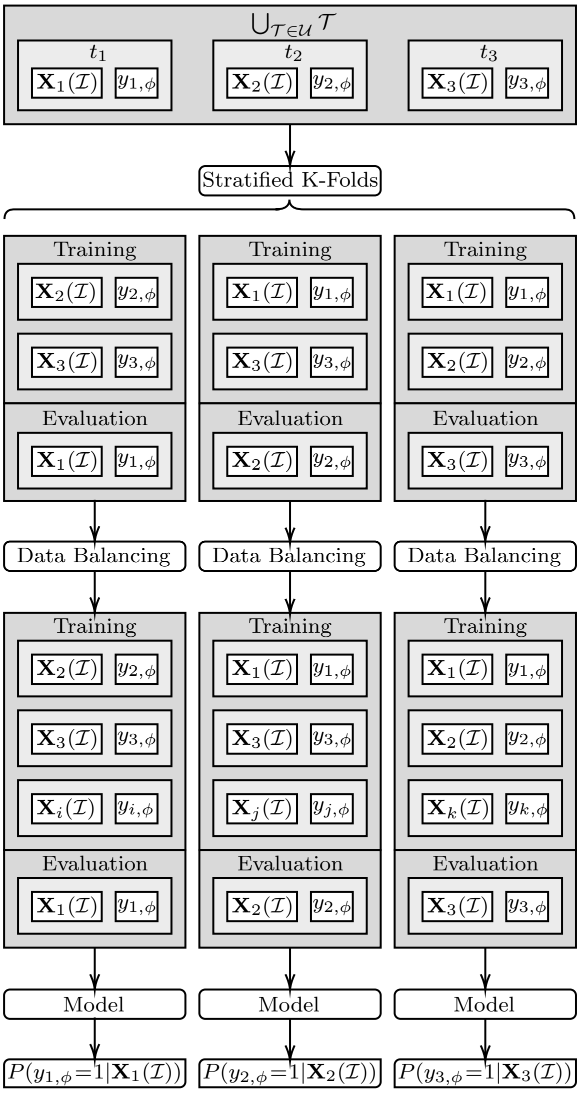

$\bigcup_{\mathcal{T}\in\mathcal{U}}\mathcal{T}$
t_1
t_2
t_3
$\mathbf{X}_1(\mathcal{I})$
$y_{1,\phi}$
$\mathbf{X}_2(\mathcal{I})$
$y_{2,\phi}$
$\mathbf{X}_3(\mathcal{I})$
$y_{3,\phi}$
Stratified K-Folds
Training
Evaluation
Data Balancing
$\mathbf{X}_i(\mathcal{I})$
$y_{i,\phi}$
$\mathbf{X}_j(\mathcal{I})$
$y_{j,\phi}$
$\mathbf{X}_k(\mathcal{I})$
$y_{k,\phi}$
Model
$P(y_{1,\phi}=1|\mathbf{X}_1(\mathcal{I}))$
$P(y_{2,\phi}=1|\mathbf{X}_2(\mathcal{I}))$
$P(y_{3,\phi}=1|\mathbf{X}_3(\mathcal{I}))$

Fig. 3: CANNIER-Framework performs stratified k-folds cross validation upon the set of all test cases in the subject set, $\bigcup_{\mathcal{T}\in\mathcal{U}}\mathcal{T}$. Following this, it applies a data balancing technique to the training portion of each fold. The framework then trains a machine learning model using the mean feature vectors $\mathbf{X}_t(\mathcal{I})$, and ground-truth labels $y_{t,\phi}$, of every test case t in each training portion. Finally, for each fold, CANNIER-Framework applies the trained model to the feature vectors of every test case in the evaluation portion. Since the evaluation portion of each fold is unique, every test case ends up with a predicted probability of being in the positive class, $P(y_{t,\phi} = 1|\mathbf{X}_t(\mathcal{I}))$.

each test case in the evaluation set being of the positive class. Since the evaluation portion of every fold is unique, after ten folds each test case in the whole subject set has a prediction. Fig. 3 offers an overview of the general pipeline. Given a lower-threshold ω_l and an upper-threshold ω_u on the predicted probability as further inputs, CANNIER-Framework assigns a *predicted label* $z_{t,\phi}(\mathcal{I}, \omega_l, \omega_u)$, to every test case, as previously shown in Fig. 1. The following equation defines the predicted label for a test case, denoted $z_{t,\phi}$.

$$z_{t,\phi}(\mathcal{I}, \omega_l, \omega_u) = \begin{cases} 0 & \text{if } P(y_{t,\phi} = 1|\mathbf{X}_t(\mathcal{I})) < \omega_l \\ 1 & \text{if } P(y_{t,\phi} = 1|\mathbf{X}_t(\mathcal{I})) \geq \omega_u \\ y_{t,\phi} & \text{if } \omega_l \leq P(y_{t,\phi} = 1|\mathbf{X}_t(\mathcal{I})) < \omega_u \end{cases} \tag{4}$$

Using the ground-truth and predicted labels for each test case in a given test suite $\mathcal{T}$, the framework calculates the frequencies of the four confusion matrix categories: true-positive (TP), false-positive (FP), false-negative (FN), and true-negative (TN). From these, it calculates the Matthews correlation coefficient (MCC) to assess the detection performance of the machine learning model for a given problem ϕ. The possible values of MCC are the closed real range between -1 and 1, where 1 indicates a model with perfect agreement between the ground-truth labels and the predicted labels and 0 indicates a model that is no better than random guessing of the predicted labels. A model with an MCC of -1 indicates perfect disagreement between the ground-truth labels and the predicted labels, such that taking a model with an MCC of 1 and inverting the predicted labels would yield an MCC of -1. We selected MCC as the overall performance metric, as opposed to F1 score, because it only produces a high value if the model performs well in terms of all four confusion matrix categories, whereas F1 score ignores true-negatives [32]. See Fig. 4 for a summary of how CANNIER-Framework combines pytest-CANNIER and the general machine learning pipeline from Fig. 3 to produce this data. The following equation defines $MCC_{\mathcal{T}_\phi}$ with respect to the four confusion matrix categories respectively denoted as $TP_{\mathcal{T}_\phi}$, $FP_{\mathcal{T}_\phi}$, $FN_{\mathcal{T}_\phi}$, and $TN_{\mathcal{T}_\phi}$.

$$\begin{aligned} TP_{\mathcal{T}_\phi}(\mathcal{I}, \omega_l, \omega_u) &= \sum_{t \in \mathcal{T}_\phi} y_{t,\phi}\, z_{t,\phi}(\mathcal{I}, \omega_l, \omega_u), \\ FP_{\mathcal{T}_\phi}(\mathcal{I}, \omega_l, \omega_u) &= \sum_{t \in \mathcal{T}_\phi} [1 - y_{t,\phi}]\, z_{t,\phi}(\mathcal{I}, \omega_l, \omega_u), \\ FN_{\mathcal{T}_\phi}(\mathcal{I}, \omega_l, \omega_u) &= \sum_{t \in \mathcal{T}_\phi} y_{t,\phi}\, [1 - z_{t,\phi}(\mathcal{I}, \omega_l, \omega_u)], \\ TN_{\mathcal{T}_\phi}(\mathcal{I}, \omega_l, \omega_u) &= \sum_{t \in \mathcal{T}_\phi} [1 - y_{t,\phi}]\, [1 - z_{t,\phi}(\mathcal{I}, \omega_l, \omega_u)], \\ MCC_{\mathcal{T}_\phi}(\mathcal{I}, \omega_l, \omega_u) &= \frac{TP_{\mathcal{T}_\phi} TN_{\mathcal{T}_\phi} - FP_{\mathcal{T}_\phi} FN_{\mathcal{T}_\phi}}{\sqrt{(TP_{\mathcal{T}_\phi} + FP_{\mathcal{T}_\phi})(TP_{\mathcal{T}_\phi} + FN_{\mathcal{T}_\phi})(TN_{\mathcal{T}_\phi} + FP_{\mathcal{T}_\phi})(TN_{\mathcal{T}_\phi} + FN_{\mathcal{T}_\phi})}} \end{aligned} \tag{5}$$

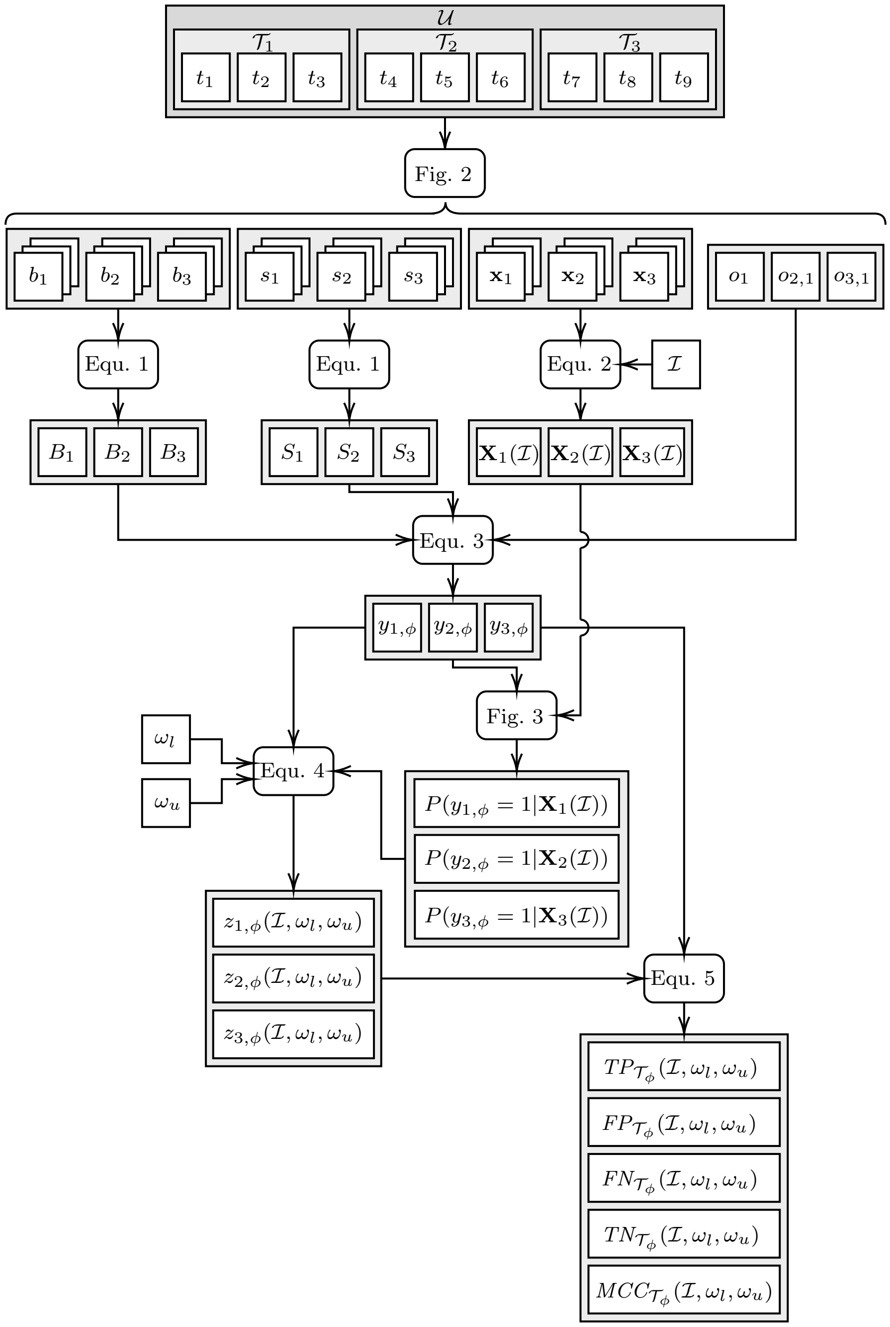

$\mathcal{U}$
$\mathcal{T}_1$
$\mathcal{T}_2$
$\mathcal{T}_3$
t_1
t_2
t_3
t_4
t_5
t_6
t_7
t_8
t_9
Fig. 2
b_1
b_2
b_3
s_1
s_2
s_3
$\mathbf{x}_1$
$\mathbf{x}_2$
$\mathbf{x}_3$
o_1
$o_{2,1}$
$o_{3,1}$
Equ. 1
Equ. 1
Equ. 2
$\mathcal{I}$
B_1
B_2
B_3
S_1
S_2
S_3
$\mathbf{X}_1(\mathcal{I})$
$\mathbf{X}_2(\mathcal{I})$
$\mathbf{X}_3(\mathcal{I})$
Equ. 3
$y_{1,\phi}$
$y_{2,\phi}$
$y_{3,\phi}$
Fig. 3
ω_l
ω_u
Equ. 4
$P(y_{1,\phi} = 1|\mathbf{X}_1(\mathcal{I}))$
$P(y_{2,\phi} = 1|\mathbf{X}_2(\mathcal{I}))$
$P(y_{3,\phi} = 1|\mathbf{X}_3(\mathcal{I}))$
$z_{1,\phi}(\mathcal{I}, \omega_l, \omega_u)$
$z_{2,\phi}(\mathcal{I}, \omega_l, \omega_u)$
$z_{3,\phi}(\mathcal{I}, \omega_l, \omega_u)$
Equ. 5
$TP_{\mathcal{T}_\phi}(\mathcal{I}, \omega_l, \omega_u)$
$FP_{\mathcal{T}_\phi}(\mathcal{I}, \omega_l, \omega_u)$
$FN_{\mathcal{T}_\phi}(\mathcal{I}, \omega_l, \omega_u)$
$TN_{\mathcal{T}_\phi}(\mathcal{I}, \omega_l, \omega_u)$
$MCC_{\mathcal{T}_\phi}(\mathcal{I}, \omega_l, \omega_u)$

Fig. 4: An overview of how CANNIER-Framework combines pytest-CANNIER and the general machine learning pipeline, with subject set $\mathcal{U}$, random sample $\mathcal{I}$, and thresholds ω_l and ω_u as input. It references previously defined figures and equations (e.g., Equ. 1 through 5 and Fig. 2 and 3).

4.2.3 Technique Evaluation Procedure

CANNIER-Framework evaluates the application of CANNIER to Rerun (CANNIER+Rerun), the Classification stage of iDFlakies (CANNIER+iDFClass), and Pairwise (CANNIER+Pairwise). We developed a mathematical model, that we implemented within the framework, to estimate the detection performance and single-core time cost associated with a set of parameters for the three techniques. CANNIER-Framework uses the ground-truth labels and predicted probabilities for each test from the NOD-vs-Rest problem ($\phi = 1$) to model CANNIER+Rerun. It uses the data from the NOD-vs-Victim problem ($\phi = 2$) to model CANNIER+iDFClass in an equivalent fashion. In both of these cases, the ground-truth labels represent the output from the "Rerun/iDFClass" block and the predicted probabilities represent the output from the "Model" block in Fig. 1a. The parameters of CANNIER+Rerun are the lower- and upper-thresholds on the model prediction, ω_l and ω_u, the sample size to produce the mean feature vectors for each test case, denoted n_F, and the maximum number of times to execute a test case without observing an inconsistent outcome, written as R_{max}. For CANNIER+iDFClass, the parameters are ω_l, ω_u, n_F, and the percentage of additional failures to recheck, denoted γ. The framework uses the outcomes from the Victim mode and the predicted probabilities from the Victim-vs-Rest ($\phi = 3$) and Polluter-vs-Rest ($\phi = 4$) problems to model CANNIER+Pairwise. The outcomes represent the "Pairwise" block and the predicted probabilities represent the "Victim model" and "Polluter model" blocks in Fig. 1b. For CANNIER+Pairwise, the parameters are the threshold for the victim model ω_V, the threshold for the polluter model ω_P, and n_F.

Given a random sample $\mathcal{I}$ of size n_F along with ω_l and ω_u, CANNIER-Framework estimates the detection performance of CANNIER+Rerun and CANNIER+iDFClass as an MCC value. For every test case t in a given test suite $\mathcal{T}$, the framework needs its individual time cost C_t, and the number of times Rerun is expected to execute it $R_t(\mathcal{I}, \omega_l, \omega_u)$, to estimate the time cost of CANNIER+Rerun, $C_{\mathcal{T}}^{Rerun}(\mathcal{I}, \omega_l, \omega_u)$. It can find C_t from the output of pytest-CANNIER in the Features mode, since this is the third feature in Table 1. As for $R_t(\mathcal{I}, \omega_l, \omega_u)$, when $P(y_{t,1} = 1|\mathbf{X}_t(\mathcal{I}))$ is not in the ambiguous region between ω_l and ω_u, CANNIER+Rerun does not delegate to Rerun and so it never executes t ($R_t(\mathcal{I}, \omega_l, \omega_u) = 0$). Otherwise, when $y_{t,1} = 0$, Rerun would execute t exactly R_{max} times since t is not NOD flaky and therefore Rerun would never observe an inconsistent outcome ($R_t(\mathcal{I}, \omega_l, \omega_u) = R_{max}$). If $y_{t,1} = 1$, Rerun would execute t until it either observes an inconsistent outcome or reaches a limit of R_{max} runs. We refer to the final run number where either of these conditions are met as r_t. In this case, $R_t(\mathcal{I}, \omega_l, \omega_u)$ is the expected value of the discrete, finite distribution $P(r_t = x)$. The probability of t giving an inconsistent outcome after exactly x runs is $\mathit{Exact}(t, x)$. This is the probability of t failing $x - 1$ times and then passing once, or passing $x - 1$ times and then failing once. When $x < R_{max}$, $P(r_t = x) = \mathit{Exact}(t, x)$. However, when $x = R_{max}$, $P(r_t = x)$ is the probability of t giving an inconsistent outcome after exactly R_{max} runs, $\mathit{Exact}(t, R_{max})$, or not giving an inconsistent outcome after reaching the limit of R_{max} runs. Where $\mathbb{E}[P]$ is the expected value of the distribution P, the definition of the time cost of CANNIER+Rerun, denoted $C_{\mathcal{T}}^{Rerun}$, is given by the following equation.

$$\begin{aligned}
C_t &= \frac{1}{N_F} \sum_{i=1}^{N_F} \mathbf{x}_{t,i,3}, \\
\mathit{Exact}(t, x) &= \left(\frac{B_t}{N_B}\right)^{x-1} \left(1 - \frac{B_t}{N_B}\right) + \left(1 - \frac{B_t}{N_B}\right)^{x-1} \frac{B_t}{N_B}, \\
P(r_t = x) &= \begin{cases} \mathit{Exact}(t, x) & \text{if } 1 < x < R_{max} \\ \mathit{Exact}(t, R_{max}) + (1 - \sum_{r=2}^{R_{max}} \mathit{Exact}(t, x)) & \text{if } x = R_{max} \end{cases}, \\
R_t(\mathcal{I}, \omega_l, \omega_u) &= \begin{cases} 0 & \text{if } P(y_{t,1} = 1 | \mathbf{X}_t(\mathcal{I})) < \omega_l \vee P(y_{t,1} = 1 | \mathbf{X}_t(\mathcal{I})) \geq \omega_u \\ R_{max} & \text{if } \omega_l \leq P(y_{t,1} = 1 | \mathbf{X}_t(\mathcal{I})) < \omega_u \wedge y_{t,1} = 0 \\ \mathbb{E}[P(r_t = x)] & \text{if } \omega_l \leq P(y_{t,1} = 1 | \mathbf{X}_t(\mathcal{I})) < \omega_u \wedge y_{t,1} = 1 \end{cases}, \\
C_{\mathcal{T}}^{Rerun}(\mathcal{I}, \omega_l, \omega_u) &= \sum_{t \in \mathcal{T}} C_t \, R_t(\mathcal{I}, \omega_l, \omega_u)
\end{aligned} \tag{6}$$

To estimate the time cost of CANNIER+iDFClass, $C_{\mathcal{T}}^{iDFClass}(\mathcal{I}, \omega_l, \omega_u)$, for a given test suite $\mathcal{T}$, CANNIER-Framework requires the number of times that iDFClass is expected to attempt to classify each test case $t \in \mathcal{T}_2$ as either NOD or a victim, $\mathit{\Gamma}_t(\mathcal{I}, \omega_l, \omega_u)$. As before, when $P(y_{t,2} = 1 | \mathbf{X}_t(\mathcal{I}, \omega_l, \omega_u))$ is not in the ambiguous region, CANNIER+iDFClass does not delegate to iDFClass and so it never classifies t ($\mathit{\Gamma}_t(\mathcal{I}, \omega_l, \omega_u) = 0$). Otherwise, iDFClass will classify a test case after its first failure during the Classification stage and will reclassify a percentage of the additional failures as determined by γ. We assume that any test case undergoing classification by iDFClass has a uniform probability of appearing at any position in the original and modified test run orders. Under this assumption, the mean length of the truncated original and modified orders would both be equal to half the size of the test suite. Therefore, the mean time cost of classifying a single test case is equal to that of one full test suite run, as given by the following equation.

$$\begin{aligned}
\mathit{\Gamma}_t(\mathcal{I}, \omega_l, \omega_u) &= \begin{cases} 0 & \text{if } P(y_{t,2} = 1 | \mathbf{X}_t(\mathcal{I})) < \omega_l \vee P(y_{t,2} = 1 | \mathbf{X}_t(\mathcal{I})) \geq \omega_u \\ 1 + \gamma(S_t - 1) & \text{if } \omega_l \leq P(y_{t,2} = 1 | \mathbf{X}_t(\mathcal{I})) < \omega_u \end{cases}, \\
C_{\mathcal{T}}^{iDFClass}(\mathcal{I}, \omega_l, \omega_u) &= \left(\sum_{t \in \mathcal{T}_2} \mathit{\Gamma}_t(\mathcal{I}, \omega_l, \omega_u) \right) \sum_{t \in \mathcal{T}} C_t
\end{aligned} \tag{7}$$

CANNIER-Framework estimates the detection performance of CANNIER+Pairwise as the ratio of victim-polluter pairs that would be detected by Pairwise to all such pairs in a given test suite $\mathcal{T}$. We selected this simpler metric, as opposed to MCC, because we assume that CANNIER+Pairwise will never incorrectly label a pair of test cases as having a victim-polluter relationship when they do not (false-positive). Under this assumption, this metric is equivalent to *true-positive rate* (TPR), also known as *sensitivity*. It has a range between 0 and 1, where 0 indicates that CANNIER+Pairwise detected none of the victim-polluter pairs and 1 indicates that it detected all of them. Since we designed the framework to only consider non-NOD flaky tests as candidate victims, such that they all have a reliable expected outcome, we have sufficient assurance that the assumption holds. Recall from Section 3.3 that CANNIER+Pairwise builds a set of victims $\mathcal{T}_V(\mathcal{I}, \omega_V)$, and polluters $\mathcal{T}_P(\mathcal{I}, \omega_P)$, given victim- and polluter-thresholds ω_V and ω_P. It then executes Pairwise with only the pairs in $\mathcal{T}_P(\mathcal{I}, \omega_P) \times \mathcal{T}_V(\mathcal{I}, \omega_V)$. CANNIER-Framework builds these sets using the predicted probabilities from the Victim-vs-Rest ($\phi = 3$) and Polluter-vs-Rest ($\phi = 4$) problems. The framework calculates TPR by dividing the number of victim-polluter pairs in $\mathcal{T}_P(\mathcal{I}, \omega_P) \times \mathcal{T}_V(\mathcal{I}, \omega_V)$ by the number of such pairs in $\mathcal{T} \times \mathcal{T}$. In other words, it divides the number of true-positives (TP) by the number of positives (P). To know how many pairs are in both sets, CANNIER-Framework relies on the outcomes recorded by pytest-CANNIER in the Victim mode. The framework estimates the time cost of CANNIER+Pairwise, $C_{\mathcal{T}}^{Pairwise}(\mathcal{I}, \omega_{\mathcal{V}}, \omega_{\mathcal{P}})$, based on the sizes of both sets and the individual time costs of their members. The definition of $TPR_{\mathcal{T}}$ and the time cost of CANNIER+Pairwise, denoted $C_{\mathcal{T}}^{Pairwise}$, is provided by the following equation.

$$\begin{aligned}
\mathcal{T}_V(\mathcal{I}, \omega_V) &= \{v | v \in \mathcal{T}, P(y_{t,3} = 1 | \mathbf{X}_t(\mathcal{I})) >= \omega_V\}, \\
\mathcal{T}_P(\mathcal{I}, \omega_P) &= \{p | p \in \mathcal{T}, P(y_{t,4} = 1 | \mathbf{X}_t(\mathcal{I})) >= \omega_P\}, \\
TP_{\mathcal{T}}(\mathcal{I}, \omega_V, \omega_P) &= \sum_{p \in \mathcal{T}_P(\mathcal{I}, \omega_P)} |\{v | v \in \mathcal{T}_V(\mathcal{I}, \omega_V) - \{p\}, o_{p,v} \neq o_v\}|, \\
P_{\mathcal{T}} &= \sum_{p \in \mathcal{T}} |\{v | v \in \mathcal{T} - \{p\}, o_{p,v} \neq o_v\}|, \\
TPR_{\mathcal{T}}(\mathcal{I}, \omega_V, \omega_P) &= \frac{TP_{\mathcal{T}}(\mathcal{I}, \omega_V, \omega_P)}{P_{\mathcal{T}}}, \\
C_{\mathcal{T}}^{Pairwise}(\mathcal{I}, \omega_V, \omega_P) &= \left(|\mathcal{T}_P(\mathcal{I}, \omega_P)| \sum_{v \in \mathcal{T}_V(\mathcal{I}, \omega_V)} C_v \right) + \left(|\mathcal{T}_V(\mathcal{I}, \omega_V)| \sum_{p \in \mathcal{T}_P(\mathcal{I}, \omega_P)} C_p \right)
\end{aligned} \tag{8}$$

5 Empirical Evaluation

We conducted experiments to answer the following research questions:

RQ1. How effective is machine learning-based flaky test detection?

RQ2. What impact do mean feature vectors have on the performance of machine learning-based flaky test detection?

RQ3. What contribution do individual features have on the output values of machine learning models for detecting flaky tests?

RQ4. What impact does CANNIER have on the performance and time cost of rerunning-based flaky test detection?

Table 3: The 30 open-source Python projects examined in this paper's study. The **Tests** column is the total number of test cases. The following three indicate the number of **NOD** flaky tests, **Victims**, and **Polluters**. The **Pairs** column gives the number of victim-polluter pairs. The **Cost** column is the combined mean time cost of every test case in seconds. The final row gives the totals for the whole subject set. Since pytest-CANNIER identifies polluters the same way as Pairwise, only considering pairs of test cases, polluters involved in more complex order-dependencies are not included. This is why the table shows that some projects appear to have victims without polluters.

GitHub Repository	**Tests**	**NOD**	**Victims**	**Polluters**	**Pairs**	**Cost (s)**
apache/airflow	3251	66	279	3241	45819	7.77×10^2
celery/celery	2332	-	15	17	24	1.31×10^2
quantumlib/Cirq	12048	-	17	2	32	8.67×10^2
conan-io/conan	3687	-	13	13	18	1.48×10^3
dask/dask	8015	1	1	37	37	1.34×10^3
encode/django-rest-framework	1402	-	1	3	3	2.63×10^3
spesmilo/electrum	542	1	1	2	2	5.99×10^1
Flexget/Flexget	1330	1	4	3	4	1.73×10^3
fonttools/fonttools	3448	1	42	-	-	1.19×10^2
graphql-python/graphene	346	-	1	1	1	1.73×10^1
facebookresearch/hydra	1538	-	19	348	952	1.77×10^2
HypothesisWorks/hypothesis	4348	5	6	3699	7401	3.92×10^3
ipython/ipython	807	6	297	796	118869	1.10×10^2
celery/kombu	1024	2	23	20	63	3.62×10^1
apache/libcloud	9809	3	133	471	1686	2.66×10^2
Delgan/loguru	1255	4	21	6	26	6.23×10^1
mitmproxy/mitmproxy	1232	-	18	338	735	3.12×10^1
python-pillow/Pillow	2567	-	26	2	26	9.38×10^1
PrefectHQ/prefect	7035	25	20	227	230	1.56×10^3
PyGithub/PyGithub	711	-	4	678	2712	5.55×10^1
Pylons/pyramid	2633	-	4	252	383	5.98×10^1
psf/requests	535	5	-	-	-	1.40×10^2
saltstack/salt	2672	12	4	65	65	2.52×10^2
scikit-image/scikit-image	6275	-	12	5882	5890	2.54×10^3
mwaskom/seaborn	1020	-	8	1	7	5.01×10^2
pypa/setuptools	694	1	23	4	4	2.08×10^2
sunpy/sunpy	1857	-	2	9	9	4.31×10^2
tornadoweb/tornado	1159	1	1	-	-	4.03×10^1
urllib3/urllib3	1320	15	1	-	-	8.57×10^1
xonsh/xonsh	4776	9	19	3114	9459	1.81×10^2
Overall	89668	158	1015	19231	194457	1.99×10^4

5.1 Subject Set

For this paper's subject set, we used the test suites of the 26 open-source Python projects studied in our previous work [55][2]. We selected these at random from a list of projects critical to open-source infrastructure created by

[2] We only reused the projects themselves as subjects. We did not reuse any of the data from our previous study.

the Open Source Security Foundation of [12]. For this paper, we randomly selected four more projects to improve the generalizability of the results. We used CANNIER-Framework to produce a dataset from these 30 test suites that contains 89,668 tests. We set the framework to perform 2,500 runs of each test suite in the Baseline mode of pytest-CANNIER ($N_B = 2500$), 2,500 runs in the Shuffle mode ($N_S = 2500$), and 30 runs in the Features mode ($N_F = 30$). Table 3 shows each project's GitHub repository; the total number of tests ($|\mathcal{T}|$); the number of NOD flaky tests ($|\mathcal{T}_1^+|$), victims ($|\mathcal{T}_3^+|$), and polluters ($|\mathcal{T}_4^+|$); the number of victim-polluter pairs ($\sum_{p \in \mathcal{T}} |\{v | v \in \mathcal{T} - \{p\}, o_{p,v} \neq o_v\}|$); and the combined mean time cost of every test in seconds ($\sum_{t \in \mathcal{T}} \big[\frac{1}{N_F} \sum_{i=1}^{N_F} \mathbf{x}_{t,i,3} \big]$).

The projects of our subject set cover a wide variety of topics. All are hosted on the Python Package Index [1] that allows developers to associate them with zero or more "topic classifiers". Topic classifiers are multi-level, for example: *Software Development :: Libraries :: Python Modules*. A developer may also specify a parent classifier on its own (e.g., just *Software Development*). Table 4 lists the topic classifiers of the 30 Python subjects. It also provides the frequencies of each classifier, taking into account their hierarchical nature.

5.2 Methodology

5.2.1 RQ1. How effective is machine learning-based flaky test detection?

The motivation behind this question is to establish a baseline for the performance of machine learning models for detecting flaky tests. While several studies have addressed this question for NOD flaky tests [22,25,55,58], and we addressed it for victims in our previous work [55], no previous study has addressed it for polluters. It is important to consider polluters when answering RQ1 since they offer developers useful information when repairing victim flaky tests and are a necessary input to techniques for mitigating them [48,53,65].

We used CANNIER-Framework to evaluate 24 concrete machine learning pipelines for each of the four flaky test classification problems. We derived these from the combination of two choices of model type, four choices of model configuration, and three choices of data balancing technique. These choices form the concrete instantiations of the "Data Balancing" and "Model" blocks in our general pipeline from Fig. 3. The two model types we considered were *random forest* [27,66] and *extra trees* [38] (the latter being a more randomized variant of the former). These are ensemble models that fit a number of decision trees [62] on subsets of the training data. We selected these particular model types due to their success in our previous work [55] and the related work of other authors [22]. The choices of model configuration were four values for the number of decision trees used by the random forest or extra trees model. These values were 25, 50, 75, and 100. In our previous work, we only considered random forest and extra trees models with 100 decision trees — the default value of our selected implementation [16]. Finally, for the three choices of data balancing, we evaluated the *synthetic minority oversampling technique* (SMOTE)

[31], SMOTE combined with *edited nearest-neighbors* (SMOTE+ENN), and SMOTE with *Tomek links* [68] (SMOTE+Tomek). SMOTE performs *oversampling*, meaning it produces synthetic data points of the minority class via interpolation. The ENN and Tomek techniques on their own perform *undersampling*, meaning they remove data points of the majority class based on similarity with their neighbors. The combination of these with SMOTE produces a hybrid balancing approach.

For each of the 24 × 4 = 96 concrete machine learning pipelines, we fixed the feature sample size at a single sample ($n_F = 1$) and had the framework repeat the model training and evaluation procedure 30 times (see Fig. 3), using a different random sample $\mathcal{I}$ to produce the mean feature vectors every time. In each instance, this resulted in 30 values of $P(y_{t,\phi} = 1)$ for every test case t and problem ϕ. To evaluate the performance of the pipelines, CANNIER-Framework needed predicted labels to calculate the confusion matrix category frequencies and MCC against the ground-truth labels for each problem. To produce the predicted labels to address this research question, we substituted $z_{t,\phi}(\mathcal{I}, \omega_l, \omega_u)$ in Equation 5 for the following definition of $z_{t,\phi}(\mathcal{I})$ that assigns a test case to its most likely class:

$$z_{t,\phi}(\mathcal{I}) = \begin{cases} 0 & \text{if } P(y_{t,\phi} = 1|\mathbf{X}_t(\mathcal{I})) < 0.5 \\ 1 & \text{if } P(y_{t,\phi} = 1|\mathbf{X}_t(\mathcal{I})) \geq 0.5 \end{cases} \tag{9}$$

With these predicted labels, we used CANNIER-Framework to calculate the confusion matrix category frequencies and the MCC of the 96 pipelines with respect to each of the 30 subject test suites in turn. We also had the framework calculate this with respect to the whole subject set for each pipeline by summing the category frequencies for each project and calculating the overall MCC from this total. This is to provide an individual assessment with respect to each test suite as well as an overview for the whole subject set. For the per-project and overall evaluations, CANNIER-Framework calculated mean values for the category frequencies and the MCC over the 30 repeats of model training and evaluation. This is to offer an evaluation that is more reliable given the non-determinism inherent to the machine learning models, the data balancing techniques, and potentially the dynamic feature values.

5.2.2 RQ2. What impact do mean feature vectors have on the performance of machine learning-based flaky test detection?

In previous studies on machine learning-based flaky test detection with dynamic test case features [22, 55], researchers performed only a single instrumented test suite run to create the feature vectors. The rationale for this question is to investigate the impact of using feature vectors that are the mean from multiple instrumented test suite runs. In the context of this study, that is multiple runs in the Features mode of pytest-CANNIER. This is to mitigate against the possible variance in the dynamic features. As an example, previous studies have found that the line coverage of test cases can vary across repeated executions [43, 63, 69]. Since three features in Table 1 are based on

Table 4: The topic classifiers of the subject projects and their frequencies. If a project declares derived classifiers we also incremented the frequencies of the parent classifiers. For a example, if a project declares *Internet*, *Internet :: Proxy Servers*, and *Software Development :: Libraries*, we would increment the frequencies of *Internet*, *Internet :: Proxy Servers*, *Software Development*, and *Software Development :: Libraries*.

Topic Classifier	**Frequency**
Communications	1
Education	1
Education :: Testing	1
Internet	5
Internet :: Proxy Servers	1
Internet :: WWW/HTTP	5
Internet :: WWW/HTTP :: WSGI	1
Multimedia	3
Multimedia :: Graphics	3
Multimedia :: Graphics :: Capture	1
Multimedia :: Graphics :: Capture :: Digital Camera	1
Multimedia :: Graphics :: Capture :: Screen Capture	1
Multimedia :: Graphics :: Graphics Conversion	2
Multimedia :: Graphics :: Viewers	1
Scientific/Engineering	4
Scientific/Engineering :: Physics	1
Scientific/Engineering :: Visualization	1
Security	1
Software Development	12
Software Development :: Build Tools	1
Software Development :: Libraries	7
Software Development :: Libraries :: Python Modules	3
Software Development :: Object Brokering	1
Software Development :: Testing	2
System	9
System :: Archiving	1
System :: Archiving :: Packaging	1
System :: Clustering	1
System :: Distributed Computing	4
System :: Logging	1
System :: Monitoring	1
System :: Networking	2
System :: Networking :: Monitoring	1
System :: Shells	1
Text Processing	1
Text Processing :: Fonts	1
Utilities	1

line coverage, we expect there to be some degree of noise in their values for each test case that could impact the detection performance of the model.

We took the best machine learning pipeline (in terms of the overall MCC) for each classification problem from the previous research question and followed the same methodology for training and evaluation, except we gave CANNIER-Framework a range of values for n_F to produce $\mathcal{I}$ between 1 and 15 samples inclusive. With 30 repeats of model training and evaluation for each value of n_F, this resulted in $15 \times 30 = 450$ rounds of stratified 10-fold cross validation for each problem. This process enabled us to investigate the correlation between the number of repeated measurements to produce the mean feature vectors and the MCC of the resultant model.

5.2.3 RQ3. What contribution do individual features have on the output values of machine learning models for detecting flaky tests?

In the interest of model explainability, we set out to investigate the impact of each individual feature in Table 1. To address this question, we applied the *Shapely Additive Explanations* (SHAP) technique [49]. It leverages concepts from game theory to quantify the contribution of an individual feature to the output value of a machine learning model for an individual data point. As inputs, SHAP takes a feature matrix and a model and returns a matrix of *SHAP values* in the same shape as the feature matrix. The SHAP value at (i, j) in the matrix represents the contribution of the jth feature on the model output for the ith data point relative to the mean output value over the dataset. This is such that summing the rows of the SHAP value matrix and adding the mean output value gives the original model output values.

In the context of this study, the features are those in Table 1, the data points are test cases, and the model output values are the predicted probabilities of each test case being in the positive class for a given flaky test classification problem. As the feature matrix, we used the mean feature vector for each test case over the 30 runs of PYTEST-CANNIER in the Features mode ($n_F = N_F$). As the machine learning model, we used CANNIER-FRAMEWORK to train the best pipeline from RQ1 using the mean feature matrix. We did this for each of the four classification problems.

Once we had a SHAP value matrix for each problem, we ranked every feature in terms of their mean absolute SHAP value over every test case. A high value would indicate that the feature has a significant impact on the model's decision (regardless of whether the impact is in favour of the negative class or the positive) and a low value would suggest the opposite. We then retrained the best pipeline for each problem with just the top 15, 12, 9, 6, and 3 features (with 30 repeats in each case). This is to observe the effect of dropping the less impactful features on the performance of the model.

5.2.4 RQ4. What impact does CANNIER *have on the performance and time cost of rerunning-based flaky test detection?*

The motivation behind this research question is to investigate if CANNIER is able to reduce the time cost of rerunning-based flaky test detection techniques while maintaining good detection performance. For the application of CANNIER to the three techniques from Section 2.1, we used CANNIER-FRAMEWORK to calculate the detection performance and single-core time cost associated with every point in a sample of their parameter spaces. For CANNIER+Rerun and CANNIER+iDFClass, the space represents the values of the 3-tuple $(\omega_l, \omega_u, n_F)$, that is, the lower-threshold, the upper-threshold, and the number of samples to produce the mean feature vectors. In the case of CANNIER+Rerun, since R_{max} (the maximum number of times to execute a test case without observing an inconsistent outcome) is a parameter of the underlying Rerun technique, rather than a parameter introduced by CANNIER, we kept its value fixed at N_B (the number of test suite runs in the Baseline mode: 2,500). Similarly, for CANNIER+iDFClass, we fixed the value of γ (the percentage of additional failures to recheck) to 20% because it is a parameter of iDFClass and not one introduced by CANNIER. This particular value was recommended by the authors of iDFlakies [47]. For the detection performance and time cost of a given point for CANNIER+Rerun/CANNIER+iDFClass, CANNIER-Framework calculated the mean over the 30 sets of predicted probabilities for the NOD-vs-Rest/NOD-vs-Victim problem from the 30 repeats of model training and evaluation for the given value of n_F from RQ2. For CANNIER+Pairwise, the parameter space represents $(\omega_V, \omega_P, n_F)$, the victim-threshold, the polluter-threshold, and the number of samples once more. In this case, the framework calculated the mean detection performance and time cost over 30 random pairs of the 30 sets of predicted probabilities for the Victim-vs-Rest problem and the 30 sets for the Polluter-vs-Rest problem for the given value of n_F.

For the sample of points in $(\omega_l, \omega_u, n_F)$, we used the values for ω_l from 0 to 1 inclusive with a step of 0.01, the values for ω_u from ω_l to 1.01 inclusive with a step of 0.01, and the values for n_F in the closed integer range from 1 to 15, except when $\omega_l = 0 \land \omega_u = 1.01$, in which case $n_F = 0$. The reason for starting from ω_l and going up to 1.01 for ω_u is to ensure that $\omega_l \leq \omega_u$ always holds and so that CANNIER-Framework evaluates the points where there is no upper-threshold on $P(y_{t,1} = 1|\mathbf{X}_t(\mathcal{I}))$ (see the second clause of Equation 4). The reason that $n_F = 0$ when $\omega_l = 0 \land \omega_u = 1.01$ is to indicate that the machine learning model, and therefore feature collection, is redundant because the ambiguous region is the entire range of $P(y_{t,1} = 1|\mathbf{X}_t(\mathcal{I}))$ under these conditions. Therefore, CANNIER+Rerun and CANNIER+iDFClass reduce to the original rerunning-based Rerun and iDFClass respectively (see the third clause of Equation 4). As the sample of points in $(\omega_V, \omega_P, n_F)$, we used the values for both ω_V and ω_P from 0 to 1 inclusive with a step of 0.01. This excludes 1.01, since when one or both thresholds is greater than 1, the set of victims and/or polluters is empty and therefore Pairwise has nothing to do since $\mathcal{T}_V(\mathcal{I}, \omega_V) \times \mathcal{T}_P(\mathcal{I}, \omega_P) = \emptyset$. For n_F, the framework considers from 1 to 15, except when $\omega_V = \omega_P = 0$, where $n_F = 0$. The reason that $n_F = 0$ in this case is to indicate that the model is redundant because $\mathcal{T}_V(\mathcal{I}, \omega_V) = \mathcal{T}_P(\mathcal{I}, \omega_P) = \mathcal{T}$ and thus CANNIER+Pairwise reduces to original Pairwise.

We had CANNIER-Framework add the time taken to collect features to the overall time cost for each point. Since many features are dynamic, they require n_F test suite runs to measure, making the time cost of doing so $n_F \sum_{t \in \mathcal{T}} C_t$ for some test suite $\mathcal{T}$. For the points where $n_F = 0$, where the other parameters render the machine learning model redundant, this additional time cost is zero. We did not consider the time cost associated with applying the model to each test case because it is negligible relative to the time taken to execute the test suite [55]. We also did not consider the time taken to train the model as part of the time cost of applying it. This is because the model only needs to be trained once and can then be applied any number of times, making training an off-line stage with a cost that can be amortized across uses.

We used CANNIER-Framework to compute the two-dimensional Pareto fronts of detection performance and time cost, with respect to the whole subject set, for the sample of points for CANNIER+Rerun, CANNIER+iDFClass, and CANNIER+Pairwise. In this context, the Pareto front represents the subset of points such that, for each point, the detection performance is the greatest compared to all other points with the same time cost. To answer this research question, we compared the detection performance and time cost associated with the point representing the balanced application of CANNIER to the point where it reduces to the original rerunning-based detection technique, for each of the three fronts. As the point representing balanced CANNIER, we used the *knee point*. The knee point is the point with the smallest Euclidean distance to the *utopia point* on the Pareto front [73]. The utopia point represents a "perfect" solution that doesn't necessarily exist. In the context of this study, that would be the point with a detection performance of 1, for either MCC or true-positive rate (TPR), and a time cost of 0 seconds. For CANNIER+Rerun and CANNIER+iDFClass, we also considered the point where they reduce to pure machine learning-based detection as an additional baseline. For this special case, we used the point on the Pareto front with the greatest MCC that also satisfies $\omega_l = \omega_u$. For all points that satisfy this condition, the techniques never defer to Rerun or iDFClass because there is no ambiguous region between the two thresholds. For CANNIER+Pairwise, there is no such point, because it only limits the problem space for Pairwise but nevertheless always defers to it.

5.3 Threats to Validity

When deciding the ground-truth labels, CANNIER-Framework could incorrectly label some flaky tests as non-flaky. We used the framework to execute every test suite 2,500 times in their original test run orders to identify NOD flaky tests and 2,500 times in shuffled orders to identify victims. Given the non-deterministic nature of flaky tests, it is generally not possible to label a test case as non-flaky with complete certainty [42]. We mitigated this issue by having CANNIER-Framework perform as many reruns as possible within the limits of our available computational resources. In total, this stage required over six weeks of computational time on a computer with a 24-core AMD Ryzen 5900X CPU. While confidence in the label increases with the number of reruns, so too does the computational cost. In our previous work [55], we found the relationship between the number of detected flaky tests and the number of test suite reruns to be sublinear. This finding supports another previous study, the authors of which identified a similar relationship [22]. This implies that continuing to re-execute a test suite gives diminishing returns with respect to the confidence of labelling a test case as non-flaky. This encourages us that the overall results of this paper would be the same had the plugin performed more reruns, because it's unlikely that it would have detected significantly more flaky tests. Furthermore, PYTEST-CANNIER is unlikely to detect certain flaky test categories by rerunning alone. For example "implementation-dependent" flaky tests may require changes to standard library implementations to manifest [64, 75]. The only category we made specific arrangements to detect were victims and their polluters; other special categories are out of the scope of this paper's empirical study.

Our concrete machine learning pipelines of the random forest/extra trees model with SMOTE data balancing and the 18 features in Table 1 may unfairly represent machine learning-based flaky test detection. A whole host of previous studies [22, 28, 29, 41, 58, 60] identified random forest to be the most suitable type of machine learning model for detecting flaky tests. In our previous work [55], we found that the extra trees model, a variant of random forest, was better suited for detecting flaky tests in some cases. Furthermore, the 18 features are based on the 16 features of FLAKE16 that we found to yield better detection performance when used to encode test cases compared to the previous state-of-the-art feature set [22]. This implies that our choice of pipeline and features is among the most suitable for detecting flaky tests currently in the literature.

There is a chance that CANNIER-FRAMEWORK and PYTEST-CANNIER contain bugs that may go on to influence the results of our evaluation. Naturally, it is impossible to be totally sure that any non-trivial software system is totally free of bugs. However, we made sure to use well-established Python libraries for the bulk of the framework's important functionality. These included COVERAGE.PY [6] to measure line coverage, PSUTIL [13] to measure many other dynamic test case properties, RADON [19] to measure source code metrics, SCIKIT-LEARN [16] for an implementation of the random forest and extra trees model, and SHAP [20] to calculate the SHAP value matrices for RQ3. 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 bespoke elements of CANNIER-FRAMEWORK and PYTEST-CANNIER.

It is possible that the results of our study would not generalize to other Python projects outside of the 30 that we sampled, or to projects written in other programming languages. We randomly sampled 30 Python projects from a list of the top-200 most critical to open-source infrastructure, as determined by the Open Source Security Foundation [12]. Part of their metric for determining the criticality of a project is based on how many other projects declare a dependency on it. Therefore, any issues caused by flaky tests in these projects could potentially impact a wider portion of the Python ecosystem. Of course, this does not guarantee that our sample generalizes to all Python projects, but does give us some assurance that the flaky tests we examined could represent a more serious problem compared to flaky tests in less critical projects. Without extending our subject set to include projects written in other languages, we cannot make any assurances that our results generalize outside of Python. Broadly speaking, however, our approach is language-agnostic. Considering Table 1, our 18 features could apply to almost any commonly used programming language. Therefore, we see no compelling reason to suggest that our results couldn't be reproduced with projects written in other languages, such as Java. In addition, it is possible that individual projects in our subject set 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 — 264% of the second highest. To resolve this concern, CANNIER-Framework calculated performance metrics with respect to each individual project.

Given the empirical nature of this paper's study, it may be difficult to reproduce our results. We took steps to make our methodology as repeatable as possible. Firstly, we included all scripts and software that we developed to facilitate this study in the replication package [4]. This includes our Dockerfile and requirements files for generating Python virtual environments [18]. Secondly, any aspects of the study that could be impacted by non-determinism, such as producing the predicted probabilities of test cases being flaky, we repeated 30 times. As such, the final results reported in this paper involve taking the mean across these 30 repeats. Finally, where any aspects of CANNIER-Framework relied on random number generators (such as when instantiating machine learning models), we made sure to set the seed to a constant value to ensure that the results are the same across repeated runs.

6 Results

6.1 RQ1. How effective is machine learning-based flaky test detection?

Table 5 shows the top-12 concrete machine learning pipelines (out of 24) for each flaky test classification problem in terms of overall MCC. Recall from Section 5.2.1 that these MCC values are with respect to the entire subject set and are the mean over 30 repeats of model training and evaluation (see Equation 5). Extra trees appears to be the best model for the NOD-vs-Rest and Victim-vs-Rest problems, and pipelines using extra trees are consistently at the top of these tables. For NOD-vs-Victim and Polluter-vs-Rest, the most performant model appears to be random forest, though with less consistency. In terms of data balancing, the best pipelines for each problem used plain SMOTE. Unlike SMOTE+Tomek, SMOTE+ENN did not make it into the top-12 for any problem. In all cases, the negative gradient of detection performance going down the table is small, such that the difference in overall MCC between the best pipeline and the 12th best pipeline is not that significant.

Tables 6 and 7 show the per-project and overall confusion matrix category frequencies (TN, FN, FP, TP) and MCC of the best pipeline for each flaky test classification problem. Table 6a shows the performance for the NOD-vs-Rest

Table 5: The top-12 pipelines (out of 24) for each flaky test classification problem in terms of overall MCC. The MCC values are the mean over 30 repeats of model training and evaluation, rounded to three significant figures.

(a) NOD-vs-Rest

Model	**Trees**	**Balancing**	**MCC**
ExtraTrees	100	SMOTE	0.532
ExtraTrees	100	+Tomek	0.529
ExtraTrees	75	SMOTE	0.527
ExtraTrees	50	SMOTE	0.526
ExtraTrees	25	SMOTE	0.521
ExtraTrees	50	+Tomek	0.519
ExtraTrees	75	+Tomek	0.519
ExtraTrees	25	+Tomek	0.507
RandomForest	100	SMOTE	0.488
RandomForest	75	SMOTE	0.479
RandomForest	50	SMOTE	0.479
RandomForest	25	SMOTE	0.477

(b) NOD-vs-Victim

Model	**Trees**	**Balancing**	**MCC**
RandomForest	75	SMOTE	0.693
RandomForest	100	SMOTE	0.690
ExtraTrees	100	SMOTE	0.689
ExtraTrees	50	SMOTE	0.686
ExtraTrees	75	SMOTE	0.686
RandomForest	50	SMOTE	0.685
RandomForest	75	+Tomek	0.684
RandomForest	25	SMOTE	0.679
RandomForest	100	+Tomek	0.675
ExtraTrees	100	+Tomek	0.673
ExtraTrees	75	+Tomek	0.669
RandomForest	25	+Tomek	0.668

(c) Victim-vs-Rest

Model	**Trees**	**Balancing**	**MCC**
ExtraTrees	75	SMOTE	0.520
ExtraTrees	100	SMOTE	0.519
ExtraTrees	50	SMOTE	0.518
ExtraTrees	100	+Tomek	0.515
ExtraTrees	75	+Tomek	0.513
ExtraTrees	50	+Tomek	0.511
ExtraTrees	25	SMOTE	0.510
ExtraTrees	25	+Tomek	0.502
RandomForest	50	SMOTE	0.501
RandomForest	75	SMOTE	0.498
RandomForest	100	SMOTE	0.498
RandomForest	25	SMOTE	0.490

(d) Polluter-vs-Rest

Model	**Trees**	**Balancing**	**MCC**
RandomForest	100	SMOTE	0.946
RandomForest	75	SMOTE	0.945
RandomForest	50	SMOTE	0.944
RandomForest	25	SMOTE	0.943
RandomForest	100	+Tomek	0.941
ExtraTrees	100	SMOTE	0.941
RandomForest	75	+Tomek	0.940
ExtraTrees	75	SMOTE	0.940
RandomForest	50	+Tomek	0.940
ExtraTrees	50	SMOTE	0.939
RandomForest	25	+Tomek	0.937
ExtraTrees	100	+Tomek	0.937

problem. The table lists relatively few projects with a defined value for MCC because many in the subject set contain zero or only very few NOD flaky tests (see Table 3). The overall MCC for this problem is 0.53 and the mean per-project MCC is close at 0.52. Recall that CANNIER-Framework calculated the overall MCC from the overall confusion matrix category frequencies, that are the sum of the per-project frequencies. An MCC of 1 indicates a perfect model and an MCC of 0 indicates a model no better than random guessing. Therefore, the detection performance of the best pipeline for this problem was fairly lackluster. Furthermore, the standard deviation of the per-project MCC is relatively high at 0.29, suggesting that the performance of the pipeline is quite variable between projects. This is further evident from the wide range of MCC values among the different projects. Table 6b shows the results for the NOD-vs-Victim problem. Once again, the table contains relatively few projects with an MCC value for the same reason as before. At 0.69, the overall MCC for this problem is greater than that for NOD-vs-Rest. Also, the standard

Table 6: The per-project and overall results of the best pipelines from Table 5 for the NOD-vs-Rest (a) and NOD-vs-Victim (b) problems. The tables give the confusion matrix category frequencies (i.e., column labels **TN**, **FN**, **FP**, **TP**), rounded to the nearest integer, and the Matthews correlation coefficient (i.e., column label **MCC**). Captions give the mean (μ) and standard deviation (σ) of the per-project MCC. Values are the mean over 30 repeats of model training and evaluation. Dashes indicate that the value is exactly zero. The "⊥" symbol indicates that the value is not defined, which was caused by a division by zero when a project does not have any test cases of certain categories.

(a) NOD-vs-Rest ($\mu = 0.52, \sigma = 0.29$)

Project	**TN**	**FN**	**FP**	**TP**	**MCC**
airflow	3159	26	26	40	0.59
celery	2332	-	-	-	⊥
Cirq	12048	-	-	-	⊥
conan	3687	-	0	-	⊥
dask	8014	1	0	-	⊥
django-rest-...	1402	-	-	-	⊥
electrum	540	1	1	-	⊥
Flexget	1329	1	-	-	⊥
fonttools	3447	1	-	-	⊥
graphene	346	-	-	-	⊥
hydra	1538	-	-	-	⊥
hypothesis	4341	5	2	-	0.00
ipython	800	2	0	4	0.76
kombu	1022	2	-	-	⊥
libcloud	9806	3	-	-	⊥
loguru	1250	2	1	2	⊥
mitmproxy	1232	-	-	-	⊥
Pillow	2567	-	0	-	⊥
prefect	7005	13	5	12	0.58
PyGithub	711	-	-	-	⊥
pyramid	2633	-	-	-	⊥
requests	530	2	-	3	0.82
salt	2660	4	0	8	0.82
scikit-image	6275	-	0	-	⊥
seaborn	1020	-	-	-	⊥
setuptools	693	1	-	-	⊥
sunpy	1857	-	-	-	⊥
tornado	1157	1	1	-	⊥
urllib3	1295	13	10	2	0.16
xonsh	4763	5	4	4	0.45
Overall	89460	83	50	75	0.53

(b) NOD-vs-Victim ($\mu = 0.55, \sigma = 0.22$)

Project	**TN**	**FN**	**FP**	**TP**	**MCC**
airflow	250	11	25	45	0.66
celery	14	-	1	-	⊥
Cirq	17	-	-	-	⊥
conan	13	-	0	-	⊥
dask	1	-	-	-	⊥
django-rest-...	0	-	1	-	⊥
electrum	0	1	1	0	⊥
Flexget	3	1	1	-	⊥
fonttools	42	-	-	-	⊥
graphene	1	-	-	-	⊥
hydra	19	-	-	-	⊥
hypothesis	6	3	0	0	⊥
ipython	296	2	1	4	0.71
kombu	23	1	-	-	⊥
libcloud	133	3	0	-	⊥
loguru	20	1	1	2	0.55
mitmproxy	6	-	0	-	⊥
Pillow	26	-	0	-	⊥
prefect	17	1	3	16	0.79
PyGithub	4	-	-	-	⊥
pyramid	4	-	-	-	⊥
requests	-	0	-	4	⊥
salt	3	1	1	11	0.69
scikit-image	12	-	-	-	⊥
seaborn	8	-	0	-	⊥
setuptools	23	0	0	1	⊥
sunpy	2	-	-	-	⊥
tornado	0	-	1	-	⊥
urllib3	-	3	1	12	0.12
xonsh	14	3	5	6	0.33
Overall	957	32	42	99	0.69

deviation of the per-project MCC is lower at 0.22. However, the mean of 0.55 is considerably lower than the overall MCC.

Table 7a gives the performance for the Victim-vs-Rest problem. At 0.52, the overall MCC is very close to the mean per-project MCC of 0.51 and is comparable to that of NOD-vs-Rest. Unlike the previous two problems, there are many more projects with a defined value for MCC, since most test suites in

Table 7: The per-project and overall results of the best pipelines from Table 5 for the Victim-vs-Rest (a) and Polluter-vs-Rest (b) problems. See Table 6 caption for more details.

(a) Victim-vs-Rest ($\mu = 0.51, \sigma = 0.24$)

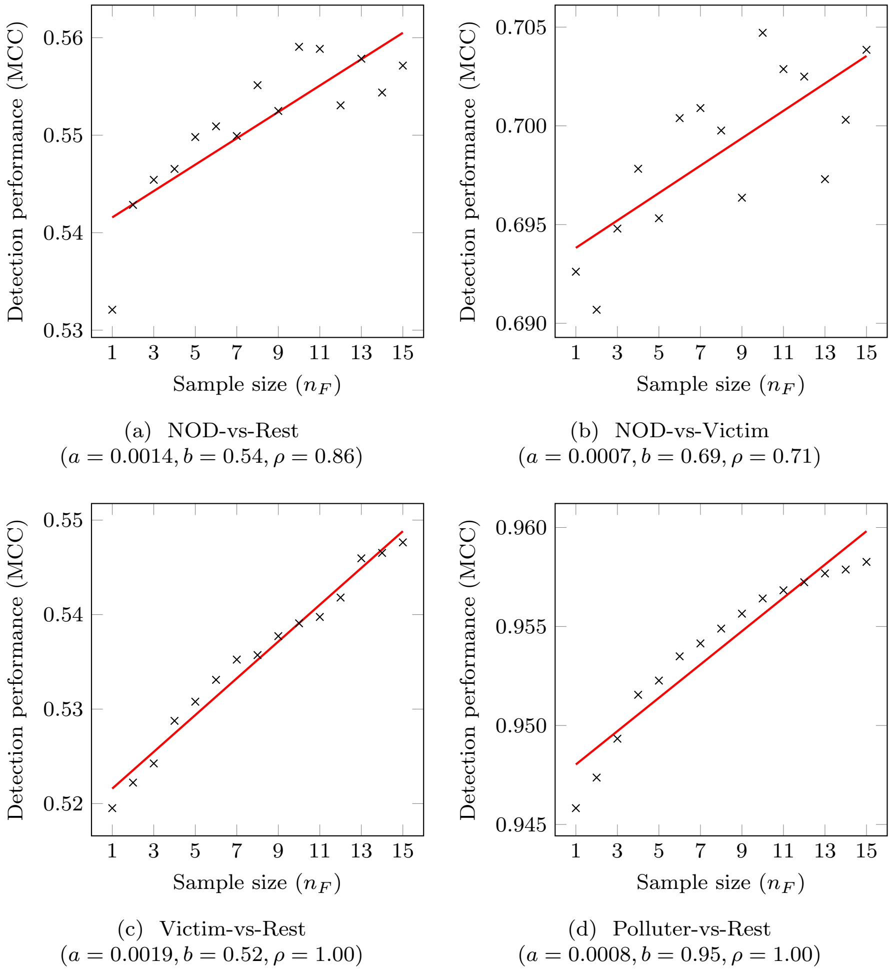
Project	**TN**	**FN**	**FP**	**TP**	**MCC**
airflow	2880	81	92	198	0.67
celery	2316	9	1	6	0.59
Cirq	12030	3	1	14	0.88
conan	3666	8	8	5	0.38
dask	8013	1	1	-	⊥
django-rest-...	1400	1	1	-	⊥
electrum	539	1	2	-	⊥
Flexget	1325	3	1	1	⊥
fonttools	3395	6	11	36	0.82
graphene	345	1	0	-	⊥
hydra	1513	13	6	6	0.37
hypothesis	4341	3	1	3	0.57
ipython	377	206	133	91	0.05
kombu	1000	12	1	11	0.68
libcloud	9612	86	64	47	0.38
loguru	1225	5	9	16	0.69
mitmproxy	1213	13	1	5	0.50
Pillow	2530	18	11	8	0.37
prefect	7014	16	1	4	0.40
PyGithub	707	1	0	3	⊥
pyramid	2629	3	0	1	⊥
requests	535	-	-	-	⊥
salt	2668	4	0	-	⊥
scikit-image	6261	5	2	7	0.69
seaborn	1009	8	3	0	⊥
setuptools	668	5	3	18	0.81
sunpy	1855	2	-	-	⊥
tornado	1156	1	2	-	0.00
urllib3	1318	1	1	-	⊥
xonsh	4753	14	4	5	0.36
Overall	88292	529	361	486	0.52

(b) Polluter-vs-Rest ($\mu = 0.46, \sigma = 0.34$)

Project	**TN**	**FN**	**FP**	**TP**	**MCC**
airflow	0	5	10	3236	0.01
celery	2311	15	4	2	0.20
Cirq	12032	1	14	1	0.12
conan	3621	6	53	7	0.25
dask	7947	0	31	37	0.73
django-rest-...	1397	3	2	-	⊥
electrum	525	2	15	-	0.01
Flexget	1327	3	0	-	⊥
fonttools	3443	-	5	-	⊥
graphene	344	1	1	-	⊥
hydra	1186	13	4	335	0.97
hypothesis	563	11	86	3688	0.91
ipython	7	165	4	631	0.13
kombu	1002	15	2	5	0.42
libcloud	9315	195	23	276	0.73
loguru	1249	6	0	0	⊥
mitmproxy	880	71	14	267	0.82
Pillow	2534	2	31	0	0.02
prefect	6781	184	27	43	0.33
PyGithub	17	8	16	670	0.58
pyramid	2355	52	26	200	0.82
requests	530	-	4	-	⊥
salt	2605	16	2	49	0.85
scikit-image	331	191	62	5691	0.71
seaborn	990	1	29	0	0.01
setuptools	686	2	4	2	0.39
sunpy	1835	6	13	2	0.21
tornado	1156	-	3	-	⊥
urllib3	1310	-	10	-	⊥
xonsh	1608	102	54	3012	0.93
Overall	69889	1077	548	18154	0.95

the subject set contained victim flaky tests. Finally, Table 7b gives the results for the Polluter-vs-Rest problem. While the overall MCC is very high at 0.95, the mean per-project MCC is much lower at 0.46 and the standard deviation is the greatest of all four problems at 0.34.

Conclusion for RQ1. The overall MCC of the best pipelines for the four classification problems ranges from 0.95 for Polluter-vs-Rest to 0.52 for Victim-vs-Rest. For NOD-vs-Rest and Victim-vs-Rest, the mean per-project MCC is close to the overall MCC. The standard deviation of the per-project MCC ranges from 0.22 to 0.34. These findings suggest that the performance of machine learning-based flaky test detection is lackluster and variable between projects, motivating the need for an alternative approach.



(a) NOD-vs-Rest
($a = 0.0014, b = 0.54, \rho = 0.86$)

(b) NOD-vs-Victim
($a = 0.0007, b = 0.69, \rho = 0.71$)

(c) Victim-vs-Rest
($a = 0.0019, b = 0.52, \rho = 1.00$)

(d) Polluter-vs-Rest
($a = 0.0008, b = 0.95, \rho = 1.00$)

Fig. 5: Plots showing that the relationship between the number of samples to produce the mean feature vectors (n_F) and the overall detection performance (MCC) of the best machine learning pipeline is positive but variable in terms of strength and gradient across the four problems. MCC values are the mean over 30 repeats. Captions give the coefficients of the red least-squares best-fit line ($MCC = a \times n_F + b$) and the Spearman's rank correlation coefficient (ρ).

6.2 RQ2. What impact do mean feature vectors have on the performance of machine learning-based flaky test detection?

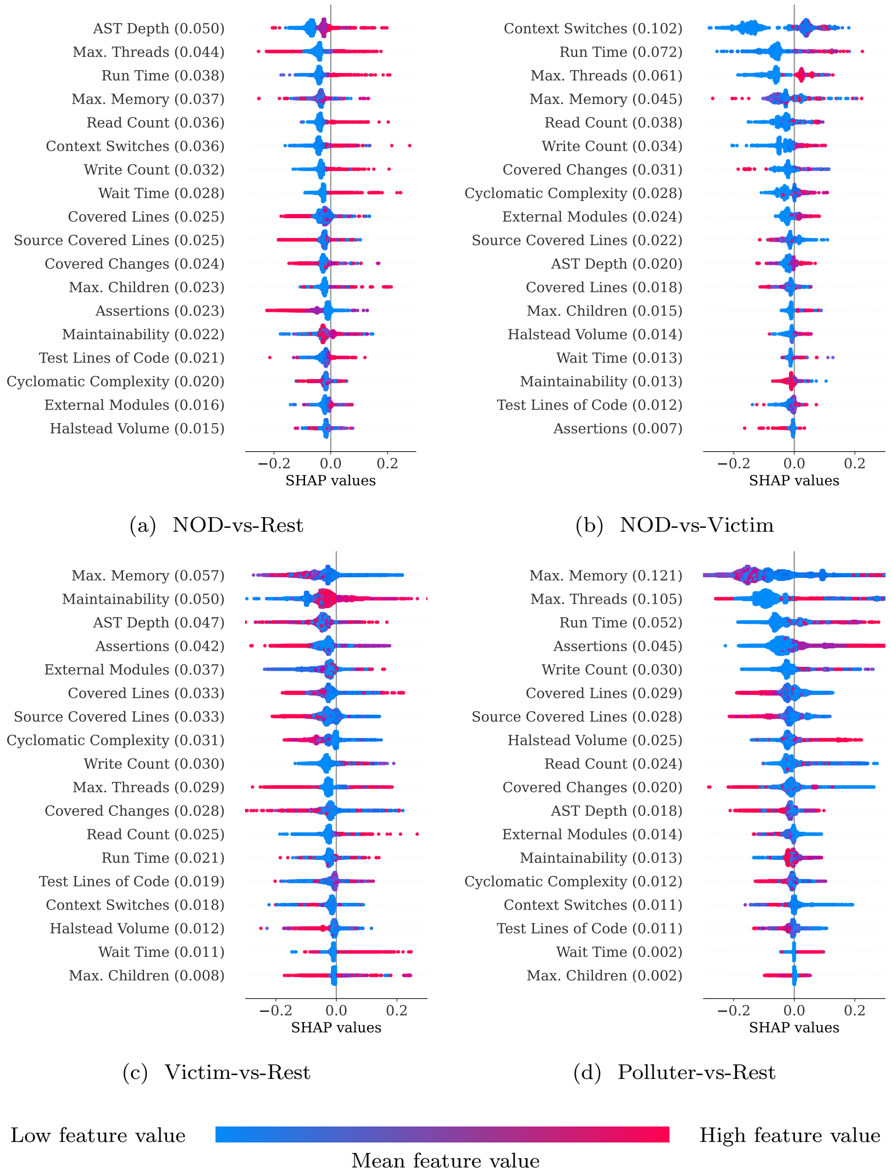
Fig. 5 shows the relationship between the sample size to produce the mean feature vectors (n_F) and the overall detection performance (MCC) of the best pipeline for each classification problem. Recall from Section 4.2.2 that CANNIER-Framework encoded test cases with feature vectors that were the mean of a random sample ($\mathcal{I}$) of the output from 30 test suite runs in the Features mode of PYTEST-CANNIER. Fig. 5a shows the relationship for the NOD-vs-Rest problem. At 0.86, the Spearman's rank correlation coefficient (ρ) indicates that the relationship is positive. However, the gradient (a) of the line of best fit (in red) is small at just 0.0014. The MCC when $n_F = 15$ on the line of best fit is only 4% greater than the MCC when $n_F = 1$. For the NOD-vs-Victim problem, Fig. 5b indicates that the relationship is weaker with a correlation coefficient of 0.71. In this case, the gradient is even smaller (0.0007), with just a 1% increase in MCC from $n_F = 15$ to $n_F = 1$.

Fig. 5c shows the relationship for Victim-vs-Rest. The correlation coefficient of 1.00 indicates a very strong positive correlation, as is clear from the plot. The gradient of the line of best fit is comparable to NOD-vs-Rest (0.0019). In the case of the Polluter-vs-Rest problem, Fig. 5d also shows a very strong positive relationship between n_F and MCC with a corresponding correlation coefficient of 1. However, the gradient is very small (0.0008).

Conclusion for RQ2. The relationship between the sample size to produce the mean feature vectors and the overall MCC of the best pipeline is positive but of variable strength across the four flaky test classification problems. For all problems, particularly NOD-vs-Victim and Polluter-vs-Rest, the gradient is small. These results indicate that using mean feature vectors has a small but positive impact on detection performance.

6.3 RQ3. What contribution do individual features have on the output values of machine learning models for detecting flaky tests?

Fig. 6 shows the SHAP values for the four flaky test classification problems as beeswarm plots. In each plot, every feature in Table 1 is represented by a row, with each value in its corresponding column in the SHAP value matrix plotted as a colored dot, for which there is one for every test case in the whole subject set. The horizontal position of each dot represents the SHAP value itself, with negative SHAP values towards the left and positive SHAP values towards the right, as indicated by the x-axis labels. Recall from Section 5.2.3 that a positive SHAP value means the contribution of the feature to the model output value from the best pipeline for a given test case and problem was positive (increased it). Conversely, a negative SHAP value means the contribution was negative (decreased it). In this context, the output value is the predicted probability of the test case belonging to the positive class of the problem. This means if a feature contributes positively to the output, it "pushes" the model towards predicting the positive class, and if it contributes negatively, it pushes towards the negative class. The color of the dots represent the feature value relative to the mean feature value, with lower values colored blue and higher values colored red. For example, a blue dot on the left side of the x-axis indicates a test case with a relatively low feature value and a positive contribution. The vertical positions of the dots represent density, such that dots with similar



(a) NOD-vs-Rest

(b) NOD-vs-Victim

(c) Victim-vs-Rest

(d) Polluter-vs-Rest

Fig. 6: SHAP values for the four flaky test classification problems as beeswarm plots. These are based on the models from best pipelines for each problem from RQ1. Blue dots represent lower feature values and red dots represent higher feature values. Purple dots represent feature values closer to the mean value. The vertical positions of the dots represent density, such that dots with similar SHAP values “swarm” around one another. Features are in descending order of their mean absolute SHAP value, which each beeswarm plot gives in parentheses. This is a measure of their overall impact on the model’s decision.

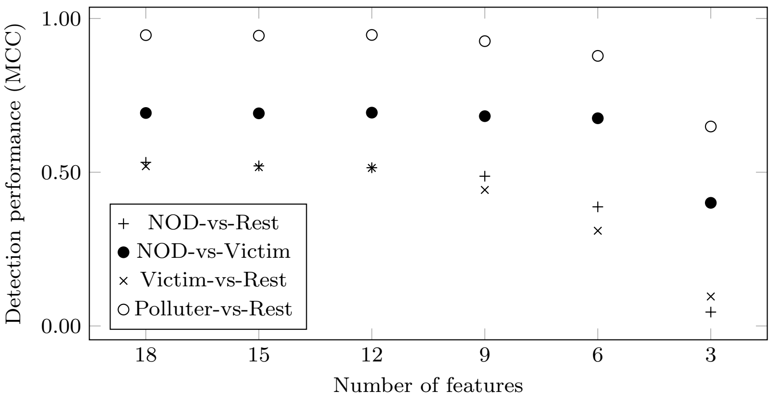


Fig. 7: The relationship between the overall MCC of the best machine learning pipelines for each flaky test classification problem and the number of top features used by CANNIER-Framework to train the model in terms of mean absolute SHAP value. On the left side of the plot, only the less impactful features are removed, which has little effect on detection performance. Towards the right, the more impactful features are dropped, resulting in a significant reduction of MCC. MCC values are the mean over 30 repeats of model training and evaluation.

SHAP values "swarm" around one another. From top-to-bottom, the features are in descending order of mean absolute SHAP value. In other words, the features closer to the top have a greater overall impact on the model output.

For the NOD-vs-Rest problem, the contribution of AST Depth, Run Time, Read Count, Context Switches, Write Count, Wait Time, Max. Children, and Test Lines of Code appears positive (towards predicting NOD flaky) when their values are high and negative when their values are low. This is evident from how the dots on the left side of their rows in Fig. 6a are mostly blue and those on the right are mostly red. Conversely, the contribution of Assertions appears negative when high and positive when low, as visualized by mostly red dots on the left and mostly blue on the right. The contribution of some features appears more nuanced. For example, when the contribution of Covered Change is negative its value is mostly high. However, when its contribution is positive its value is mixed. For the NOD-vs-Victim problem (Fig. 6b), Context Switches, Run Time, Max. Threads, Read Count, Write Count, Cyclomatic Complexity, External Modules, Max. Children, and Halstead Volume appear to contribute positively (towards predicting NOD flaky) when their values are high and negatively when low. The contribution of the individual features for this problem appear considerably less well-defined compared to NOD-vs-Rest. There are some similarities between the results for these two problems, such as Run Time, Read Count, Context Switches, Write Count, and Max. Children mostly contributing positively when high and negatively when low.

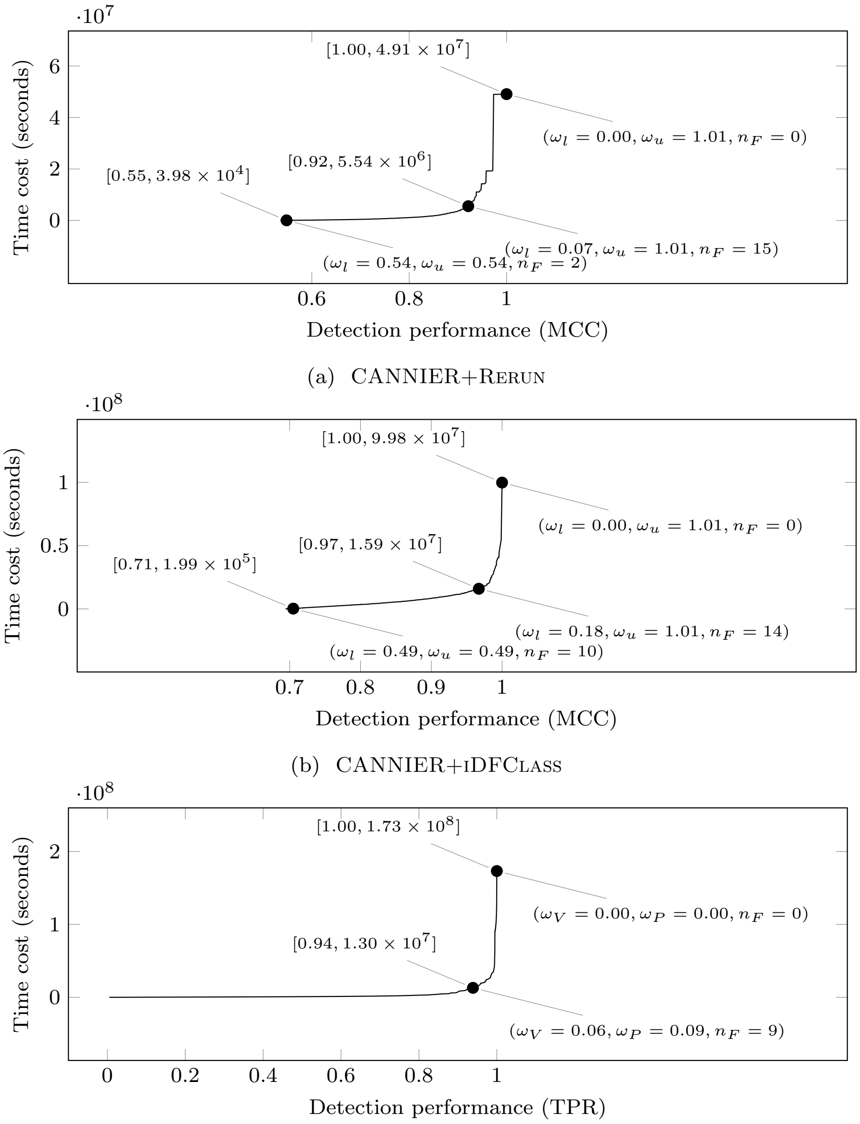
As shown by Fig. 6c the contribution of Maintainability, Write Count, Read Count, and Wait Time features appear broadly positive when their values are high (towards predicting victim flaky) and negative when low. On the other hand, Source Covered Lines, Cyclomatic Complexity, and Halstead Volume show the opposite behavior with moderate consistency. The impact of the features for this problem differs significantly compared to the NOD-vs-Victim problem. For example, the Maintainability and Cyclomatic Complexity feature appears to have nearly the exact opposite contribution pattern. Finally, for the Polluter-vs-Rest problem (Fig. 6d), Run Time, Assertions, Halstead Volume, and Wait Time contribute positively when high. Covered Lines, Source Covered Lines, and Max. Children show the opposite contribution.

Figure 7 shows how the overall MCC of the best pipelines for each problem decreases as the number of features used by CANNIER-Framework to train the model are reduced, starting from the least impactful in terms of mean absolute SHAP value. For example, for the NOD-vs-Rest problem, the MCC value at 6 on the x-axis corresponds to a model that only considers AST Depth, Max. Threads, Run Time, Max. Memory, Read Count, and Context Switches. Initially, the detriment to detection performance is fairly small as only the least important features are pruned. However, at around 9 features, the overall MCC begins to plummet quite considerably for every problem.

Conclusion for RQ3. In terms of mean absolute SHAP value, the most impactful features for the NOD-vs-Rest problem were AST Depth, Max. Threads, and Run Time; and for the NOD-vs-Victim problem were Context Switches, Run Time, and Max. Threads. For the Victim-vs-Rest problem, the most impactful were Max. Memory, Maintainability, and AST Depth; and for the Polluter-vs-Rest problem were Max. Memory, Max. Threads, and Run Time. Some features had a clear contribution pattern to the model and others less so, suggesting potentially more complex relationships.

6.4 RQ4. What impact does CANNIER have on the performance and time cost of rerunning-based flaky test detection?

Fig. 8 shows the Pareto fronts of overall detection performance and time cost for the application of CANNIER to the three rerunning-based detection techniques (see Equations 5, 6, 7, and 8). From right-to-left, the first pin on each curve is at the point representing the original rerunning-based technique (where the machine learning model becomes redundant). The second is at the point representing the balanced application of CANNIER (the knee point). Tables 8, 9, and 10 give the per-project and overall results at this point. For CANNIER+Rerun and CANNIER+iDFClass, the third is at the point representing pure machine learning-based detection (greatest MCC where $\omega_l = \omega_u$). Above each pin in square brackets is the detection performance and time cost associated with the point (its coordinates on the axes). Below in parentheses are its parameters.



(a) CANNIER+Rerun

(b) CANNIER+iDFClass

(c) CANNIER+Pairwise

Fig. 8: The Pareto fronts of detection performance and time cost for the application of CANNIER to the three rerunning-based detection techniques. From right-to-left, the first pin on each curve is at the point representing the original rerunning-based technique. The second is at the point representing the balanced application of CANNIER. For CANNIER+Rerun (a) and CANNIER+iDFClass (b), the third is at the point representing pure machine learning-based detection. There is no third pin for CANNIER+Pairwise (c) because it is not possible to use a pure machine learning-based approach in this context (see Section 5.2.4). Above each pin in square brackets is the detection performance and time cost with respect to the whole subject set. Below in parentheses are the parameters.

Table 8: The per-project and overall results for CANNIER+Rerun. The table gives the confusion matrix categories, rounded to the nearest integer, and the MCC at the point in the parameter space representing balanced CANNIER+Rerun ($\omega_l = 0.07, \omega_u = 1.01, n_F = 15$). It also gives the time cost (in seconds) at this point (CANNIER+) and at the point representing original Rerun (**Original**). The time cost is significantly reduced when using CANNIER. Values are the mean over 30 repeats of model training and evaluation. Dashes indicate that the value is exactly zero. The "⊥" symbol indicates that the value is not defined, which was caused by a division by zero when a project does not have any test cases of certain categories.

						Time cost (s)	
Project	**TN**	**FN**	**FP**	**TP**	**MCC**	CANNIER+	**Original**
airflow	3185	3	-	63	0.98	7.71×10^5	1.69×10^6
celery	2332	-	-	-	⊥	8.53×10^4	3.28×10^5
Cirq	12048	-	-	-	⊥	2.32×10^5	2.17×10^6
conan	3687	-	-	-	⊥	2.42×10^5	3.70×10^6
dask	8014	-	-	1	1.00	3.01×10^5	3.34×10^6
django-rest-...	1402	-	-	-	⊥	8.85×10^4	6.57×10^6
electrum	541	1	-	-	⊥	5.12×10^4	1.39×10^5
Flexget	1329	1	-	-	⊥	7.25×10^4	4.32×10^6
fonttools	3447	1	-	-	⊥	1.09×10^4	2.97×10^5
graphene	346	-	-	-	⊥	2.55×10^3	4.31×10^4
hydra	1538	-	-	-	⊥	2.34×10^4	4.42×10^5
hypothesis	4343	4	-	1	⊥	1.68×10^6	9.57×10^6
ipython	801	0	-	6	0.98	5.04×10^4	2.63×10^5
kombu	1022	1	-	1	⊥	1.27×10^4	9.05×10^4
libcloud	9806	1	-	2	⊥	5.00×10^3	6.66×10^5
loguru	1251	2	-	2	⊥	4.73×10^4	1.48×10^5
mitmproxy	1232	-	-	-	⊥	2.11×10^3	7.79×10^4
Pillow	2567	-	-	-	⊥	2.89×10^4	2.35×10^5
prefect	7010	2	-	23	0.96	4.05×10^5	3.80×10^6
PyGithub	711	-	-	-	⊥	1.41×10^4	1.39×10^5
pyramid	2633	-	-	-	⊥	1.10×10^3	1.49×10^5
requests	530	0	-	5	1.00	8.48×10^4	3.46×10^5
salt	2660	1	-	11	0.96	8.75×10^4	6.27×10^5
scikit-image	6275	-	-	-	⊥	5.73×10^5	6.36×10^6
seaborn	1020	-	-	-	⊥	6.41×10^4	1.25×10^6
setuptools	693	1	-	-	⊥	1.40×10^5	4.75×10^5
sunpy	1857	-	-	-	⊥	3.89×10^5	1.08×10^6
tornado	1158	1	-	-	⊥	6.44×10^3	1.01×10^5
urllib3	1305	3	-	12	0.89	3.53×10^4	2.13×10^5
xonsh	4767	1	-	8	0.94	2.94×10^4	4.50×10^5
Overall	89510	24	-	134	0.92	5.54×10^6	4.91×10^7

Fig. 8a and Table 8a give the results for CANNIER+Rerun. As shown by the figure, the time cost associated with the point representing balanced CANNIER+Rerun (middle pin) is 89% lower than the time cost associated with the point representing original Rerun (right pin). At 0.92, the MCC at the balanced CANNIER+Rerun point is significantly greater than the

Table 9: The per-project and overall results for CANNIER+iDFClass. The table gives the confusion matrix categories, rounded to the nearest integer, and the MCC at the point in the parameter space representing balanced CANNIER+iDFClass ($\omega_l = 0.18, \omega_u = 1.01, n_F = 14$). It also gives the time cost (in seconds) at this point (CANNIER+) and at the point representing original iDFClass (**Original**). See Table 8 caption for more details.

						Time cost (s)	
Project	**TN**	**FN**	**FP**	**TP**	**MCC**	CANNIER+	**Original**
airflow	275	3	-	53	0.97	9.22×10^6	7.24×10^7
celery	15	-	-	-	$\perp$	6.95×10^4	2.06×10^5
Cirq	17	-	-	-	$\perp$	3.23×10^4	4.47×10^6
conan	13	-	-	-	$\perp$	2.47×10^4	4.84×10^5
dask	1	-	-	-	$\perp$	1.87×10^4	6.60×10^5
django-rest-...	1	-	-	-	$\perp$	1.00×10^6	9.66×10^5
electrum	1	1	-	0	$\perp$	1.11×10^3	3.95×10^2
Flexget	4	1	-	0	$\perp$	1.44×10^6	1.75×10^6
fonttools	42	-	-	-	$\perp$	2.91×10^4	2.48×10^6
graphene	1	-	-	-	$\perp$	2.42×10^2	4.27×10^3
hydra	19	-	-	-	$\perp$	2.93×10^4	1.31×10^6
hypothesis	6	1	-	2	0.85	6.32×10^5	9.55×10^5
ipython	297	0	-	5	0.99	1.80×10^4	7.84×10^5
kombu	23	1	-	-	$\perp$	4.63×10^3	1.38×10^5
libcloud	133	1	-	2	$\perp$	8.73×10^4	8.74×10^6
loguru	21	0	-	3	0.98	4.15×10^4	2.35×10^5
mitmproxy	6	-	-	-	$\perp$	2.61×10^3	3.70×10^4
Pillow	26	-	-	-	$\perp$	2.55×10^3	1.51×10^4
prefect	20	-	-	17	1.00	7.84×10^5	1.04×10^6
PyGithub	4	-	-	-	$\perp$	7.82×10^2	3.11×10^2
pyramid	4	-	-	-	$\perp$	8.37×10^2	3.54×10^4
requests	-	-	-	4	$\perp$	1.81×10^4	1.61×10^4
salt	4	-	-	12	1.00	1.26×10^6	1.33×10^6
scikit-image	12	-	-	-	$\perp$	4.89×10^5	6.98×10^5
seaborn	8	-	-	-	$\perp$	7.64×10^4	1.29×10^5
setuptools	23	-	-	1	1.00	7.53×10^4	1.95×10^5
sunpy	2	-	-	-	$\perp$	6.20×10^3	1.94×10^5
tornado	1	-	-	-	$\perp$	6.05×10^2	4.03×10^1
urllib3	1	0	-	15	0.99	1.25×10^4	1.13×10^4
xonsh	19	1	-	8	0.95	5.18×10^5	5.75×10^5
Overall	999	8	-	123	0.97	1.59×10^7	9.98×10^7

MCC at the point representing pure machine learning-based detection (left pin), which is 0.55. As shown by the table, the per-project MCC is very consistent. Naturally, the MCC at the original Rerun point is exactly 1, since the predicted labels are the same as the ground-truth labels in this case (see Equation 4). Furthermore, the time cost at the pure machine learning point is significantly lower than the time cost at the other points of interest. This is because the only time cost associated with this point is that of collecting feature data. These results demonstrate that applying CANNIER to Rerun can

Table 10: The per-project and overall results for CANNIER+Pairwise. The table gives the number of detected victim-polluter pairs (**TP**), the total number of such pairs (**P**), and the true-positive rate (**TPR**) at the point in the parameter space representing balanced CANNIER+Pairwise ($\omega_V = 0.06, \omega_P = 0.09, n_F = 9$). It also gives the time cost (in seconds) at this point (CANNIER+) and at the point representing original Pairwise (**Original**). See Table 8's caption for more details about the entities in this table.

				Time cost (s)	
Project	**TP**	**P**	**TPR**	CANNIER+	**Original**
airflow	45490	45819	0.99	2.60×10^6	5.05×10^6
celery	7	24	0.31	2.95×10^4	6.12×10^5
Cirq	30	32	0.94	1.15×10^5	2.09×10^7
conan	-	18	-	2.13×10^5	1.09×10^7
dask	1	37	0.03	2.39×10^5	2.15×10^7
django-rest-...	0	3	0.07	4.92×10^4	7.37×10^6
electrum	-	2	-	7.57×10^3	6.49×10^4
Flexget	2	4	0.43	1.70×10^4	4.60×10^6
fonttools	-	-	$\perp$	1.93×10^4	8.19×10^5
graphene	-	1	-	3.66×10^2	1.19×10^4
hydra	839	952	0.88	3.48×10^4	5.44×10^5
hypothesis	4071	7401	0.55	2.07×10^6	3.41×10^7
ipython	112497	118869	0.95	1.56×10^5	1.78×10^5
kombu	44	63	0.70	8.20×10^3	7.42×10^4
libcloud	984	1686	0.58	1.42×10^5	5.23×10^6
loguru	3	26	0.13	3.77×10^3	1.56×10^5
mitmproxy	90	735	0.12	1.13×10^4	7.68×10^4
Pillow	23	26	0.88	4.57×10^4	4.82×10^5
prefect	103	230	0.45	5.29×10^5	2.19×10^7
PyGithub	2703	2712	1.00	1.45×10^4	7.89×10^4
pyramid	262	383	0.68	4.34×10^3	3.15×10^5
requests	-	-	$\perp$	6.63×10^3	1.50×10^5
salt	50	65	0.78	2.96×10^4	1.34×10^6
scikit-image	5887	5890	1.00	6.30×10^6	3.19×10^7
seaborn	5	7	0.72	8.31×10^4	1.02×10^6
setuptools	4	4	1.00	1.76×10^4	2.89×10^5
sunpy	-	9	-	1.48×10^5	1.60×10^6
tornado	-	-	$\perp$	3.05×10^3	9.35×10^4
urllib3	-	-	$\perp$	7.90×10^3	2.26×10^5
xonsh	9442	9459	1.00	1.15×10^5	1.73×10^6
Overall	182538	194457	0.94	1.30×10^7	1.73×10^8

significantly reduce its time cost while maintaining a detection performance that is far greater than the extra trees model alone.

Fig. 8b and Table 8b show the results for CANNIER+iDFClass. As shown by the figure, the general picture is similar to CANNIER+Rerun but somewhat attenuated. The reduction in time cost from original iDFClass to balanced CANNIER+iDFClass is 84%, slightly less than that for CANNIER+Rerun. In addition, the difference in MCC between the balanced CANNIER+iDFClass point (0.97) and the pure machine learning point

(0.71) is slightly less significant. The per-project MCC is broadly consistent, as shown by the table. The overall implications of these results are the same as before, namely that applying CANNIER to iDFClass scarifies a minimal degree of detection performance for a considerable reduction in time cost.

Fig. 8c and Table 8c give the results for CANNIER+Pairwise. Again, the overall story is similar to the two prior techniques. In this case, the drop in time cost between original Pairwise and balanced CANNIER+Pairwise is the greatest at 92%. Furthermore, the true-positive rate (TPR) at the point representing balanced CANNIER+Pairwise is very high at 0.94. Yet, the table shows that the per-project detection performance varies significantly, far more than the previous two techniques. This could be explained by the relatively high variance in the per-project detection performance of the machine learning pipeline for the Polluter-vs-Rest problem (see Table 7b).

Conclusion for RQ4. When applied to Rerun, iDFClass, and Pairwise, CANNIER is able to reduce time cost by an average of 88% at the expense of only a minor decrease in detection performance.

7 Discussion

7.1 RQ1. How effective is machine learning-based flaky test detection?

As shown by Table 5, there is not much difference in terms of overall MCC between consecutive pipelines in the top-12 for each classification problem. Nonetheless, there are some patterns that have emerged from our choice of pipeline configurations. For NOD-vs-Rest and Victim-vs-Rest, it appears that extra trees is the clear winner for the type of model, consistently occupying the top positions in both tables. Extra trees is a more randomized variant of random forest, an ensemble model based on decision trees [27,38,62,66]. Both fit individual trees on a random subset of the features from a random sample of the data points from the training data. The major difference between the two models is how nodes in the decision tree are split. Random forest uses an optimal split, whereas extra trees uses a random split. The additional randomness introduced by extra trees trades increased *bias* for reduced *variance*. Increased bias means the model may fail to recognize relationships between feature data and labels, known as *underfitting*. Reduced variance means the model may be less sensitive to noise and outliers, avoiding *overfitting*. The fact that extra trees was more performant with respect to NOD-vs-Rest and Victim-vs-Rest could suggest that this particular trade-off was more beneficial when tackling these two problems, compared to NOD-vs-Victim and Polluter-vs-Rest. The reason for this however would require further investigation.

The pipelines with more trees tended to yield greater detection performance than those of the same model type and balancing but with fewer trees. This is expected, since the motivation behind random forest and extra trees is to fit decision trees with decoupled prediction errors, such that taking an average of their individual predictions leads to some errors cancelling out. Therefore, it stands to reason that more trees would lead to greater performance. Of course, increasing the number of trees can only improve the model up to a point — and, moreover, there are some instances in our results where more trees did not lead to better performance.

Plain SMOTE (without additional underbalancing) appeared to yield better pipelines compared to SMOTE+ENN and SMOTE+Tomek. Recall from Section 5.2.1 that SMOTE [31] synthetically increases the number of data points in the minority class via interpolation. However, the combination of SMOTE with additional underbalancing techniques produces both synthetic members of the minority class but also discards some members of the majority class. It could be that the removal of real data points was detrimental to the performance of the pipelines that used these techniques, though further investigation would be required to be sure.

Table 6b shows the per-project and overall results of the best pipeline for the NOD-vs-Victim problem. There is a fairly significant difference between the overall MCC of 0.69 and the per-project mean MCC of 0.55. Recall that CANNIER-Framework calculates the overall MCC from the sum of the per-project confusion matrix category frequencies. This disparity is probably caused by the individual results for IPython and Airflow having a disproportionate impact on the overall result since they have significantly more victim flaky tests than the other subject projects (see Table 3). This is also seen in Table 7b for Polluter-vs-Rest, though in this case the difference between the mean and overall MCC is much larger. Once again, this is likely due to the influence of individual projects with relatively many polluters.

The per-project MCC varies quite considerably, with a standard deviation ranging from 0.22 to 0.34 across the four problems. We would expect that projects with fewer flaky tests would have a poorer MCC than those with more, simply because they have fewer positive examples to train the model. However, our results do not appear to show this trend. Therefore, further investigation is required to fully understand why the MCC for some projects is so much greater than that of others.

7.2 RQ2. What impact do mean feature vectors have on the performance of machine learning-based flaky test detection?

Our conclusion for RQ2, as illustrated by Fig. 5, is that increasing the sample size to produce the mean feature vectors increases the overall MCC of the best pipeline for the four flaky test classification problems. This is not surprising, given how the literature has already established a degree of non-determinism in some of the dynamic features in Table 1 [43,63,69]. What is more interesting is how weak the effect on MCC appears to be, despite being clearly positive, as illustrated by the very small gradient of the line of best fit. Despite this, at the point representing balanced CANNIER for all three flaky test detection techniques in RQ4, the number of samples to produce the mean feature vectors (n_F) is fairly high (15, 14, and 9 for CANNIER+Rerun, CANNIER+iDFClass, and CANNIER+Rerun, respectively). This suggests that the added time cost of performing the extra feature measurements may be a worthwhile trade-off for the increased detection performance.

7.3 RQ3. What contribution do individual features have on the output values of machine learning models for detecting flaky tests?

Fig. 6 gives the SHAP value beeswarm plots based on the best pipelines for the four flaky test classification problems. These visualize the contribution of the 18 features in Table 1 towards the output value of the model for a given test case. It is important to remember that random forest and extra trees are not causal models and therefore it is not appropriate to infer causality by applying SHAP without considering confounding [33]. Furthermore, as demonstrated by our results for RQ1, the detection performance of the models is limited and therefore the SHAP values may not even offer a reliable insight into the correlations between the feature values and the probability of a test case being flaky. Despite this, some of our findings support general intuition and the consensus of the flaky test literature.

For the NOD-vs-Rest problem, we found that Wait Time appears to contribute positively to the extra trees model output (towards predicting NOD flaky) when its value is high and negatively when low. This feature measures the elapsed wall-clock time spent waiting for input/output (I/O) operations to complete. Many empirical studies have pointed to "asynchronous waiting" as a leading cause of NOD flaky tests [35,46,50,61], where a test case waits for an insufficient amount of time for an asynchronous operation, such as I/O, to complete. We also found Context Switches and Max. Children to have a similar contribution pattern. Both of these features are associated with concurrency, another leading cause of flakiness as attested by the same studies. Furthermore, Read Count and Write Count, that measure the number of times the filesystem performed input and output respectively, also appear to contribute positively to the model output when high and negatively when low. Previous work has identified I/O itself as a cause of flaky tests [50], but this behavior could also be related to asynchronous waiting, since Wait Time is time spent waiting for I/O and could correlated with Read Count and Write Count.

For NOD-vs-Rest and NOD-vs-Victim, Run Time has a positive contribution when high and a negative contribution when low and ranks highly in terms of overall contribution (i.e., the mean absolute SHAP value). In their evaluation of FlakeFlagger, Alshammari et al. [22] also found the execution time of test cases to be correlated with the probability of being NOD flaky. However, they were unable to establish any casual link. For the Victim-vs-Rest problem, Write Count, Read Count, and Wait Time seem to contribute have a similar contribution pattern, but to varying degrees of consistency. Since these features are associated with I/O, this correlation could be explained by the

```
def test_create_pool(self):
    pool = self.client.create_pool(name='foo', slots=1, description='')
    self.assertEqual(pool, ('foo', 1, ''))
    self.assertEqual(self.session.query(models.Pool).count(), 2)
```

(a) This test case from the Airflow project [2] has an AST depth of 1.

```
def test_top_level_return_error(self):
    tl_err_test_cases = self._get_top_level_cases()
    tl_err_test_cases.extend(self._get_ry_syntax_errors())

    vals = ('return', 'yield', 'yield from (_ for _ in range(3))',
            dedent('''
                def f():
                    pass
                return
                '''),
            )

    for test_name, test_case in tl_err_test_cases:
        # This example should work if 'pass' is used as the value
        with self.subTest((test_name, 'pass')):
            iprc(test_case.format(val='pass'))

        # It should fail with all the values
        for val in vals:
            with self.subTest((test_name, val)):
                msg = "Syntax error not raised for %s, %s" % (test_name, val)
                with self.assertRaises(SyntaxError, msg=msg):
                    iprc(test_case.format(val=val))
```

(b) This test case from the IPython project [10] has an AST depth of 5.

Fig. 9: Two test cases with different values for the AST depth feature. This feature measures the maximum depth of nested program statements.

relationship between filesystem activity and victim flaky tests established in previous studies (e.g., [23,26,36,50,76]).

Seven of the 18 features are static, meaning they are based on the test case code and do not require a test case execution to measure. One of these is AST Depth that measures the maximum depth of nested program statements. Fig. 9 compares two test cases with different values for the AST depth feature. In terms of mean absolute SHAP value, AST Depth was the most impactful for the NOD-vs-Rest problem. While no previous study has examined the relationship between AST Depth and flakiness, intuitively we might expect a high AST Depth to be associated with a higher chance of flakiness. This is simply because a test case with a higher AST Depth is likely to be more complex and therefore offer more opportunities for flakiness to arise. The beeswarm plot for NOD-vs-Rest appear to broadly support this notion yet the plots for the other problems do not indicate a clear relationship. This suggests that AST Depth may be correlated with the probability of a test case being NOD flaky.

There appear to be some tentative relationships between the contribution patterns of features for the four problems. For NOD-vs-Rest and NOD-vs-Victim, the contribution of Run Time, Read Count, Context Switches, Write Count, and Max. Children are broadly positive when high and negative when low. This could be due to the positive class being the same for both problems and the negative class of NOD-vs-Victim being a subset of the negative class of NOD-vs-Rest. Moreover, the contribution pattern of the features for the NOD-vs-Victim differs significantly that of the Victim-vs-Rest problem. As we saw in Section 6.3, the Maintainability and Cyclomatic Complexity features appear to have nearly opposite contribution patterns between the two problems. This is expected, because the positive class of Victim-vs-Rest is the negative of NOD-vs-Victim, and the negative class of Victim-vs-Rest is a superset of the positive of NOD-vs-Victim.

It is clear from Figure 7 that dropping the less impactful features (in terms of mean absolute SHAP value) has little impact on the detection performance of the best pipeline for each problem. Since the time to fit a random forest/extra trees model grows linearly with the number of features, this is a useful result for expediting the training stage. This is not directly relevant to the conclusions of this paper's study however, as we are not concerned with the time cost of model training since that is performed off-line from the perspective of a developer using the CANNIER approach.

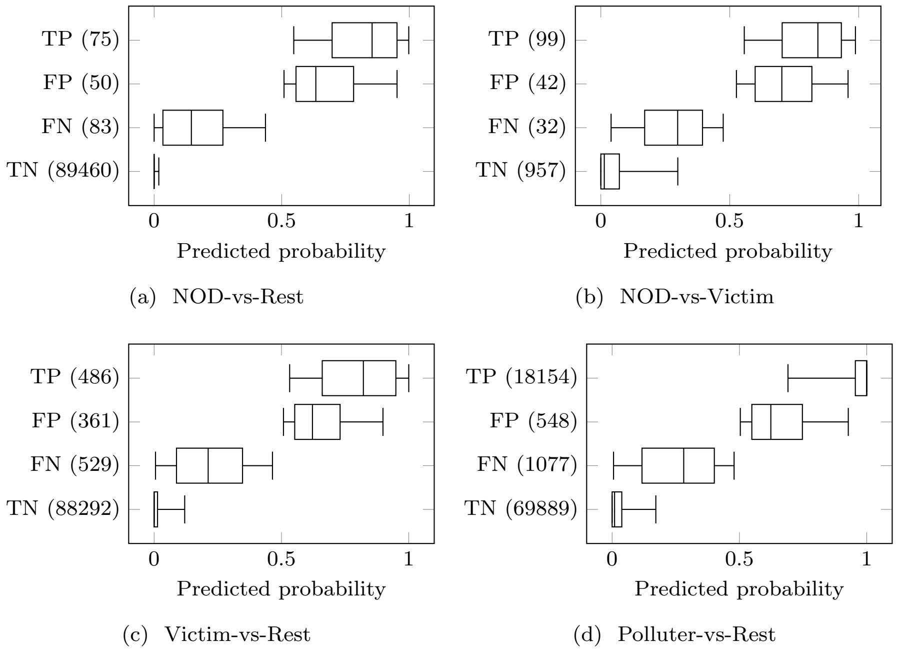
7.4 RQ4. What impact does CANNIER have on the performance and time cost of rerunning-based flaky test detection?

We presented CANNIER+iDFClass as a drop-in replacement for the Classification stage of iDFlakies. In theory, the combination of the NOD-vs-Rest and Victim-vs-Rest models could be a substitute for the entire iDFlakies pipeline. This could be realized as CANNIER+iDFlakies, a multi-model approach with a multi-label output: NOD, Victim, or Rest (non-flaky). In practice, the difficulty arises when either of the models are ambiguous for a given test case. To delegate the prediction for such a test case to iDFlakies in this hypothetical scenario, CANNIER+iDFlakies would need to rerun the entire test suite in different orders until the test case fails or the upper-limit is reached. This corresponds to the Running stage of iDFlakies. As with the single-model CANNIER+iDFClass given in the paper, it would then execute the prefix of the failing test order, representing the Classification stage of iDFlakies. Naturally, with even a handful of ambiguous cases, the hypothetical multi-model CANNIER+iDFlakies would be unlikely to noticeably reduce the time cost of the Running stage, but would reduce the time cost of the Classification stage in the same way as the existing single-model CANNIER+iDFClass. Therefore, the benefit of CANNIER+iDFlakies is effectively the same as CANNIER+iDFClass, since the latter makes no attempt to expedite the Running stage. For these reasons, we opted to focus on CANNIER+iDFClass due to its simplicity and the fact that it would require fewer modifications to iDFlakies to implement.

As shown in Fig. 8a and Table 8, for the point representing balanced CANNIER+Rerun, the lower-threshold (ω_l) is very low at 0.07 and the upper-threshold (ω_u) is at its maximum value of 1.01. The latter means that there effectively is no upper-threshold on the predicted probability (see the second clause of Equation 4). Fig. 10 illustrates the distribution of predicted probabilities for test cases in, and gives the frequencies of, each confusion matrix category, for each of the four flaky test classification problems. We produced this figure from the results of RQ1, such that the figure for each classification problem corresponds to its respective table in Tables 6 and 7. Fig. 10a focuses on the NOD-vs-Rest problem. The distribution for true-negatives (TN) is focused largely around 0 and represents the vast majority of test cases. Furthermore, the distribution for false-negatives (FN) appears highly separable from true-negatives. This might explain why ω_l is so low, because it means CANNIER+Rerun labels most true-negative test cases as negative and prevents them from being delegated to Rerun, significantly reducing time cost. It also means CANNIER+Rerun labels only a handful of false-negatives as negative, limiting the reduction in detection performance. The distribution for true-positives (TP) is clearly different from false-positives (FP) but not as easily separable. However, there are few test cases in both categories relative to true-negatives. Therefore, by setting ω_u to its maximum value, CANNIER+Rerun makes no false-positive predictions, ensuring no decrease in detection performance at the expense of a minor increase in time cost. This could explain why there are no false-positive predictions in Table 8.

Fig. 10b illustrates the distribution of predicted probabilities for NOD-vs-Victim. The situation for this problem and the thresholds for the point representing balanced CANNIER+iDFClass is very similar to NOD-vs-Rest and CANNIER+Rerun. The biggest difference is that the frequency of the true-negative category for NOD-vs-Victim is two orders of magnitude smaller than that for NOD-vs-Rest. The distribution for true-negatives also spreads much further into the distribution for false-negatives. This may explain why the lower-threshold for CANNIER+iDFClass is greater at 0.18 and why the reduction in time cost from iDFClass to CANNIER+iDFClass is smaller.

Fig. 10c and 10d are for Victim-vs-Rest and Polluter-vs-Rest respectively. Once again, the overall picture is similar for both problems. That is, the true-negative category contains the vast majority of test cases and its distribution is broadly separable from the false-negative category. This explains why the victim-threshold (ω_V) and polluter-threshold (ω_P) for the balanced CANNIER+Pairwise point are low at 0.06 and 0.08 respectively. Uniquely for Polluter-vs-Rest, the true-positive distribution appears very distinct from the false-positive distribution. Perhaps because this problem has significantly more positive examples in the dataset compared to the other problems, the machine learning model can discern unseen positive cases with greater confidence.



(a) NOD-vs-Rest

(b) NOD-vs-Victim

(c) Victim-vs-Rest

(d) Polluter-vs-Rest

Fig. 10: The distribution of predicted probabilities for test cases in, and the frequencies of, each confusion matrix category, for each of the four flaky test classification problems. The data is based on the best pipelines from RQ1. Whiskers represent the range from the 5th to the 95th percentile and boxes represent the 25th to the 75th. Middle lines represent the median (50th).

7.5 Implications

7.5.1 Researchers

Our findings for RQ1 extend the existing body of work in machine learning-based flaky test detection into the detection of polluter test cases. Identifying polluters is vital for mitigating test-order dependencies [48,53,65] and so our results demonstrate the wider applicability of machine learning models for tackling flaky tests. Our results for RQ2 (and supported by RQ4) demonstrate that using mean feature vectors can improve the detection performance of machine learning models. We therefore suggest that researchers consider the implications of this when evaluating machine-learning based techniques that use dynamic features. Our results for RQ3 tentatively identify correlations between test case metrics and the probability of a test case being flaky. This is an important foundation for future work in elevating flaky test detection techniques to comprehensive flaky test root causing techniques, a vital intermediate step towards automated flaky test repair. While such root causing and repair techniques exist [45,67,70], they are expensive and limited in scope.

7.5.2 Developers

Our findings for RQ4 demonstrate that CANNIER is a “best of both worlds” approach between rerunning-based and machine learning-based flaky test detection. As shown by Figure 8, CANNIER reduces time cost by an average of 88% across the three rerunning-based techniques while maintaining good detection performance. For developers, this means not having to trade high time cost for limited detection performance. Furthermore, while we used the knee-point of the Pareto front to represent CANNIER in our evaluation, developers could customize the approach towards lower time cost or greater detection performance by selecting a different point.

8 Related Work

Luo et al. [50] performed one of the earliest empirical studies of test flakiness. Using 51 projects of the Apache Software Foundation as subjects, they classified 201 commits that repaired flaky tests into 10 categories based on the cause of the flakiness. The most common cause they identified was related to waiting for asynchronous operations. For example, a test case that launches a thread to perform input/output (I/O) and waits a fixed amount of time for it to finish may fail when it takes longer than expected. One of our findings for RQ3 was that the amount of time spent waiting for I/O operations to complete was positively correlated with the probability of a test case being NOD flaky.

Gruber et al. [40] repeatedly executed the test suites of 22,352 open-source projects and automatically identified 7,571 flaky tests. Like our study, these projects were primarily written in the Python programming language. They randomly sampled 100 NOD flaky tests in their dataset to classify their causes using the categories introduced by Luo et al. [50]. Unlike Luo et al., they found causes related to networking and randomness to be the most prevalent.

Bell et al. [24] presented an automated technique, called DeFlaker, for detecting NOD flaky tests. The key advantage of DeFlaker over Rerun is that it does not require repeated test case executions. Instead, the technique takes advantage of a project’s history in a version control system. When a test case that passed on a previous version of the software now fails, and does not cover modified code, DeFlaker labels it as flaky. Naturally, DeFlaker requires a test suite run with code instrumentation to measure coverage. Detecting flaky tests using extra trees models with CANNIER-Framework also requires an instrumented run to measure coverage and the other metrics in Table 1. In both cases, this test suite run introduces time overhead. However, DeFlaker requires a run every time a change is made, whereas CANNIER-Framework requires at least one to produce encodings for each test case that would likely remain relevant over a series of changes. Furthermore, DeFlaker can only detect flaky tests after they fail. In contrast, the models trained by the CANNIER-Framework can detect flaky tests preemptively.

Pinto et al. [58] and Bertolino et al. [25] both presented machine learning-based flaky test detection techniques based purely on static features of the test case code. Both techniques encoded test cases using a bag-of-words approach. This represents 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. used additional static features such as the number of lines of code. Bertolino et al. used a *k-nearest neighbor* classifier [44] for the machine learning model and Pinto et al. evaluated a range of different models, including random forest. They found random forest to yield the best detection performance, of which we use the extra trees variant in this paper's study, having found it to be the most effective in our prior work [55]. Alshammari et al. [22] presented FlakeFlagger, a detection technique using a random forest model and encoding test cases with a feature set containing a mixture of static and dynamic test case metrics. 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 cost of a single instrumented test suite run to measure the dynamic features. For this reason, we included both static and dynamic test metrics in our feature set instead of relying on purely static features.

Shi et al. [65] presented iFixFlakies, a technique for automatically generating patches for victim flaky tests. Their approach uses delta-debugging [74] to identify a victim's polluters and other test cases that may contain the statements needed to repair the victim, known as *cleaners*. CANNIER+Pairwise could provide a drop-in replacement for this aspect of iFixFlakies. However, we cannot say for certain if our approach would be faster than using delta-debugging because we have not yet evaluated it in this context.

Lam et al. [47] presented iDFlakies, a technique for detecting flaky tests and classifying them as either NOD or Victim. The overall process involves repeatedly executing a test suite in a modified order (e.g., shuffled) to identify flaky test cases. Following this, the tool enters a Classification stage where it attempts to determine the category of each flaky test. In this paper's study, we evaluated the application of CANNIER to the Classification stage of this tool (CANNIER+iDFClass). Our empirical results demonstrated that CANNIER was able to significantly reduce the execution time overhead of the Classification stage at minimal detriment to its detection performance.

9 Conclusion and Future Work

This paper expanded the existing work on machine learning-based flaky test detection and introduced CANNIER, an approach for significantly reducing the time cost of rerunning-based detection techniques by combining them with machine learning models. Initially, using a variety of machine learning pipelines and a feature set of 18 static and dynamic test case metrics, we performed a baseline evaluation of machine learning-based detection on our dataset of 89,668 test cases from 30 Python projects. We evaluated their performance with respect to detecting NOD flaky tests, victim flaky tests, and polluter test cases. Our results suggested that the performance of the machine learning models was lackluster and variable between projects. We then went on to investigate the impact of mean feature vectors on machine learning-based flaky test detection. We identified a positive relationship between the sample size to produce the mean feature vectors and the detection performance of the machine learning model. In the interest of model explainability, we applied the SHAP technique [49] to quantify the contribution of each individual feature to the output value of the model. While this technique can only reveal correlations and is not appropriate for inferring causality, we made several findings that support both the general intuition of developers and results from the flaky test literature. Finally, we evaluated CANNIER's impact on three rerunning-based methods for flaky test detection Rerun, the Classification stage of iDFlakies, and Pairwise. We found that CANNIER was able to significantly reduce time cost at the expense of only a minor decrease in detection performance.

As future work, we intend to further investigate the features associated with test flakiness. In doing so, we will consider applying causal inference techniques [72] for a deeper understanding into the processes that lead to test flakiness. We will also consider evaluating the performance of machine-learning-based detection with respect to more specific categories of flaky tests, such as "implementation-dependent" flaky tests [64,75]. Finally, we plan to evaluate the efficiency and effectiveness of integrating CANNIER into a wider range of existing flaky test techniques, such as iFixFlakies [65].

10 Data Availability Statement

The datasets generated during and/or analyzed during this paper's study are available in the CANNIER-Experiment repository, `https://github.com/flake-it/cannier-experiment`.

11 Declarations

The research leading to these results received funding from the Engineering and Physical Sciences Research Council (EPSRC) (Award Number: EP/R513313/1).

References

1. (2022)
2. airflow/test_local_client.py at c743b95a02ba1ec04013635a56ad042ce98823d2, `https://github.com/apache/airflow/blob/c743b95a02ba1ec04013635a56ad042ce98823d2/tests/api/client/test_local_client.py#L127` (2022)
3. apache/airflow at c743b95a02ba1ec04013635a56ad042ce98823d2 `https://github.com/apache/airflow/tree/c743b95a02ba1ec04013635a56ad042ce98823d2` (2022)
4. CANNIER experiment, `https://github.com/flake-it/cannier-experiment` (2022)
5. CANNIER framework, `https://github.com/flake-it/cannier-framework` (2022)

6. Coverage.py — Coverage.py 6.4.1 documentation, https://coverage.readthedocs.io/en/stable/ (2022)
7. Docker documentation, https://docs.docker.com/ (2022)
8. Glossary — Python 3.10.4 documenation, https://docs.python.org/3/glossary.html\#term-global-interpreter-lock (2022)
9. I/O statistics fields, https://www.kernel.org/doc/Documentation/iostats.txt (2022)
10. ipython/test_async_helpers.py at 95d2b79a2bd889da7a29e7c3cf5f49c1d25ff43d, https://github.com/ipython/ipython/blob/95d2b79a2bd889da7a29e7c3cf5f49c1d25ff43d/IPython/core/tests/test_async_helpers.py#L135 (2022)
11. New EC2 M5zn instances — Fastest Intel Xeon scalable CPU in the cloud — AWS news blog https://aws.amazon.com/blogs/aws/new-ec2-m5zn-instances-fastest-intel-xeon-scalable-cpu-in-the-cloud/ (2022)
12. Open source project criticality score (beta), https://github.com/ossf/criticality_score (2022)
13. Psutil documentation — Psutil 5.7.3 documenation, https://psutil.readthedocs.io/en/stable/ (2022)
14. pytest-CANNIER, https://github.com/flake-it/pytest-cannier (2022)
15. Pytest: Helps you write better programs — Pytest documentation, https://docs.pytest.org/en/7.1.x/ (2022)
16. Scikit-learn: Machine learning in Python — Scikit-learn 1.1.1 documenation, https://scikit-learn.org/stable/ (2022)
17. Unittest — Unit testing framework — Python 3.10.4 documenation, https://docs.python.org/3/library/unittest.html (2022)
18. Virtual environments and packages — Python 3.10.4 documenation, https://docs.python.org/3/tutorial/venv.html (2022)
19. Welcome to radon's documenation! — Radon 4.1.0 documenation https://radon.readthedocs.io/en/stable/index.html (2022)
20. Welcome to the SHAP documenation! — SHAP latest documenation https://shap.readthedocs.io/en/stable/index.html (2022)
21. Al-Qutaish, R., Abran, A.: Halstead Metrics: Analysis of their Design, pp. 145–159. Wiley (2010)
22. Alshammari, A., Morris, C., Hilton, M., Bell, J.: FlakeFlagger: Predicting flakiness without rerunning tests. In: Proceedings of the International Conference on Software Engineering (ICSE) (2021)
23. Bell, J., Kaiser, G., Melski, E., Dattatreya, M.: 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), pp. 770–781 (2015)
24. Bell, J., Legunsen, O., Hilton, M., Eloussi, L., Yung, T., Marinov, D.: DeFlaker: Automatically detecting flaky tests. In: Proceedings of the International Conference on Software Engineering (ICSE), pp. 433–444 (2018)
25. Bertolino, A., Cruciani, E., Miranda, B., Verdecchia, R.: Know your neighbor: Fast static prediction of test flakiness. IEEE Access **9**, 76119–76134 (2021)
26. Biagiola, M., Stocco, A., Mesbah, A., Ricca, F., Tonella, P.: 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), pp. 154–164 (2019)
27. Breiman, L.: Random forests. Machine Learning **45**(1), 5–32 (2001)
28. Camara, B., Silva, M., Endo, A., S., V.: On the use of test smells for prediction of flaky tests. In: Proceedings of the Brazilian Symposium on Systematic and Automated Software Testing (SAST), pp. 46–54 (2021)
29. Camara, B., Silva, M., Endo, A., S., V.: What is the vocabulary of flaky tests? An extended replication. In: Proceedings of the International Conference on Program Comprehension (ICPC), pp. 444–454 (2021)
30. Candido, J., Melo, L., D'Amorim, M.: 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), pp. 153–158 (2017)

31. Chawla, N.V., Bowyer, K.W., Hall, L.O., Kegelmeyer, W.P.: SMOTE: Synthetic minority over-sampling technique. Journal of Artificial Intelligence Research **16**, 321–357 (2002)
32. Chicco D. Jurman, G.: The advantages of the matthews correlation coefficient (mcc) over f1 score and accuracy in binary classification evaluation. BMC Genomics **21**(6), 1471–2164 (2020)
33. Dillon, E., LaRiviere, J., Lundberg, S., Roth, J., Syrgkanis, V.: Be careful when interpreting predictive models in search of causal insights, `https://shap.readthedocs.io/en/latest/example_notebooks/overviews/Be%20careful%20when%20interpreting%20predictive%20models%20in%20search%20of%20causal%C2%A0insights.html` (2018)
34. Durieux, T., Goues, C.L., Hilton, M., Abreu, R.: Empirical study of restarted and flaky builds on Travis CI. In: Proceedings of the International Conference on Mining Software Repositories (MSR), pp. 254–264 (2020)
35. Eck, M., Palomba, F., Castelluccio, M., Bacchelli, A.: 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), pp. 830–840 (2019)
36. Gambi, A., Bell, J., Zeller, A.: Practical test dependency detection. In: Proceedings of the International Conference on Software Testing, Verification and Validation (ICST), pp. 1–11 (2018)
37. Garousi, V., Küçük, B.: Smells in software test code: A survey of knowledge in industry and academia. Journal of Systems and Software **138**, 52–81 (2018)
38. Geurts, P., Ernst, D., Wehenkel, L.: Extremely randomized trees. Machine Learning **63**(1), 3–42 (2006)
39. Gill, G.K., Kemerer, C.F.: Cyclomatic complexity density and software maintenance productivity. Transactions on Software Engineering **17**(12), 1284 (1991)
40. Gruber, M., Lukasczyk, S., Kroiß, F., Fraser, G.: An empirical study of flaky tests in Python. In: Proceedings of the International Conference on Software Testing, Verification and Validation (ICST) (2021)
41. Haben, G., Habchi, S., Papadakis, M., Cordy, M., Le Traon, Y.: 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)
42. Harman, M., O'Hearn, P.: 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), pp. 1–23 (2018)
43. Hilton, M., Bell, J., Marinov, D.: A large-scale study of test coverage evolution. In: Proceedings of the International Conference on Automated Software Engineering (ASE), pp. 53–63 (2018)
44. Keller, J.M., Gray, M.R., Givens, J.A.: A fuzzy k-nearest neighbor algorithm. Transactions on Systems, Man, and Cybernetics **15**(4), 580–585 (1985)
45. Lam, W., Godefroid, P., Nath, S., Santhiar, A., Thummalapenta, S.: Root causing flaky tests in a large-scale industrial setting. In: Proceedings of the International Symposium on Software Testing and Analysis (ISSTA), pp. 204–215 (2019)
46. Lam, W., Muşlu, K., Sajnani, H., Thummalapenta, S.: A study on the lifecycle of flaky tests. In: Proceedings of the International Conference on Software Engineering (ICSE), pp. 1471–1482 (2020)
47. Lam, W., Oei, R., Shi, A., Marinov, D., Xie, T.: IDFlakies: A framework for detecting and partially classifying flaky tests. In: Proceedings of the International Conference on Software Testing, Verification and Validation (ICST), pp. 312–322 (2019)
48. Lam, W., Shi, A., Oei, R., Zhang, S., Ernst, M.D., Xie, T.: Dependent-test-aware regression testing techniques. In: Proceedings of the International Symposium on Software Testing and Analysis (ISSTA), pp. 298–311 (2020)
49. Lundberg, S.M., Erion, G., Chen, H., DeGrave, A., Prutkin, J.M., Nair, B., Katz, R., Himmelfarb, J., Bansal, N., Lee, S.: From local explanations to global understanding with explainable AI for trees. Nature Machine Intelligence **2**(1), 2522–5839 (2020)
50. Luo, Q., Hariri, F., Eloussi, L., Marinov, D.: An empirical analysis of flaky tests. In: Proceedings of the Symposium on the Foundations of Software Engineering (FSE), pp. 643–653 (2014)

51. Machalica, M., Samylkin, A., Porth, M., Chandra, S.: Predictive test selection. In: Proceedings of the International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP), pp. 91–100 (2019)
52. Memon, A., Gao, Z., Nguyen, B., Dhanda, S., Nickell, E., Siemborski, R., Micco, J.: Taming Google-scale continuous testing. In: Proceedings of the International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP), pp. 233–242 (2017)
53. Parry, O., Kapfhammer, G.M., Hilton, M., McMinn, P.: Flake it 'till you make it: Using automated repair to induce and fix latent test flakiness. In: Proceedings of the International Workshop on Automated Program Repair (APR), pp. 11–12 (2020)
54. Parry, O., Kapfhammer, G.M., Hilton, M., McMinn, P.: A survey of flaky tests. Transactions on Software Engineering and Methodology **31**(1), 1–74 (2021)
55. Parry, O., Kapfhammer, G.M., Hilton, M., McMinn, P.: Evaluating features for machine learning detection of order- and non-order-dependent flaky tests. In: Proceedings of the International Conference on Software Testing, Verification and Validation (ICST), pp. 93–104 (2022)
56. Parry, O., Kapfhammer, G.M., Hilton, M., McMinn, P.: Surveying the developer experience of flaky tests. In: Proceedings of the International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP) (2022)
57. Peitek, N., Apel, S., Parnin, C., Brechmann, A., Siegmund, J.: Program comprehension and code complexity metrics: An fMRI study. In: International Conference on Software Engineering (ICSE), pp. 524–536 (2021)
58. Pinto, G., Miranda, B., Dissanayake, S., Amorim, M.D., Treude, C., Bertolino, A., D'amorim, M.: What is the vocabulary of flaky tests? In: Proceedings of the International Conference on Mining Software Repositories (MSR), pp. 492–502 (2020)
59. Pontillo, V., Palomba, F., Ferrucci, F.: Toward static test flakiness prediction: A feasibility study. In: Proceedings of the International Workshop on Machine Learning Techniques for Software Quality Evoluton, pp. 19–24 (2021)
60. Pontillo, V., Palomba, F., Ferrucci, F.: Static test flakiness prediction: How far can we go? (2022)
61. Romano, A., Song, Z., Grandhi, S., Yang, W., Wang, W.: An empirical analysis of UI-based flaky tests. In: Proceedings of the International Conference on Software Engineering (ICSE) (2021)
62. Safavian, S.R., Landgrebe, D.: A survey of decision tree classifier methodology. Transactions on Systems, Man, and Cybernetics **21**(3), 660–674 (1991)
63. Shi, A., Bell, J., Marinov, D.: Mitigating the effects of flaky tests on mutation testing. In: Proceedings of the International Symposium on Software Testing and Analysis (ISSTA), pp. 296–306 (2019)
64. Shi, A., Gyori, A., Legunsen, O., Marinov, D.: Detecting assumptions on deterministic implementations of non-deterministic specifications. In: Proceedings of the International Conference on Software Testing, Verification and Validation (ICST), pp. 80–90 (2016)
65. Shi, A., Lam, W., Oei, R., Xie, T., Marinov, D.: 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), pp. 545–555 (2019)
66. Shi, T., Horvath, S.: Unsupervised learning with random forest predictors. Journal of Computational and Graphical Statistics **15**(1), 118–138 (2006)
67. Terragni, V., Salza, P., Ferrucci, F.: A container-based infrastructure for fuzzy-driven root causing of flaky tests. In: Proceedings of the International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER), pp. 69–72 (2020)
68. Tomek, I.: Two modifications of CNN. Transactions on Systems, Man, and Cybernetics **6**, 769–772 (1976)
69. Vysali, S., McIntosh, S., Adams, B.: Quantifying, characterizing, and mitigating flakily covered program elements. Transactions on Software Engineering (2020)
70. Wei, A., Yi, P., Li, Z., Xie, T., Marinov, D., Lam, W.: Preempting flaky tests via non-idempotent-outcome tests. In: Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS) (2022)
71. Welker, K.D.: The software maintainability index revisited. CrossTalk **14**, 18–21 (2001)

72. Yao, L., Chu, Z., Li, S., Li, Y., Gao, J., Zhang, A.: A survey on causal inference. Transactions on Knowledge Discovery from Data (TKDD) **15**(5), 1–46 (2021)
73. Zavala, V.M., Flores-Tlacuahuac, A.: Stability of multiobjective predictive control: A utopia-tracking approach. Automatica **48**(10), 2627–2632 (2012)
74. Zeller, A., Hildebrandt, R.: Simplifying and isolating failure-inducing input. Transactions on Software Engineering **28**(2), 183–200 (2002)
75. Zhang, P., Jiang, Y., Wei, A., Stodden, V., Marinov, D., Shi, A.: Domain-specific fixes for flaky tests with wrong assumptions on underdetermined specifications. In: Proceedings of the International Conference on Software Engineering (ICSE), p. 50–61 (2021)
76. Zhang, S., Jalali, D., Wuttke, J., Muşlu, K., Lam, W., Ernst, M.D., Notkin, D.: Empirically revisiting the test independence assumption. In: Proceedings of the International Symposium on Software Testing and Analysis (ISSTA), pp. 385–396 (2014)