Know Your Neighbor: Fast Static Prediction of Test Flakiness

Antonia Bertolino
ISTI - CNR
Pisa, Italy
antonia.bertolino@isti.cnr.it

Emilio Cruciani
Gran Sasso Science Institute
L'Aquila, Italy
emilio.cruciani@gssi.it

Breno Miranda*
Federal University of Pernambuco
Recife, Brazil
bafm@cin.ufpe.br

Roberto Verdecchia†
Gran Sasso Science Institute
L'Aquila, Italy
roberto.verdecchia@gssi.it

ABSTRACT

Flaky tests plague regression testing in Continuous Integration environments by slowing down change releases, wasting development effort, and also eroding testers trust in the test process. We present FLAST, the first static approach to flakiness detection using test code similarity. Our extensive evaluation on 24 projects taken from repositories used in three previous studies showed that FLAST can identify flaky tests with up to 0.98 Median and 0.92 Mean precision. For six of those projects it could already yield ~0.98 average precision values with a training set containing less than 100 tests. Besides, where known flaky tests are classified according to their causes, the same approach can also predict a flaky test category with alike precision values. The cost of the approach is negligible: the average train time over a dataset of ~1,700 test methods is less than one second, while the average prediction time for a new test is less than one millisecond.

1 INTRODUCTION

Flaky tests *can intermittently pass or fail even for the same code version* [26]. A seminal study in 2014 by Luo et al. [26] analyzed empirically the common causes and manifestations of test flakiness and brought the attention of the research community onto this serious problem, which was already well-known among practitioners, e.g., [10, 12, 40].

Flakiness hinders regression testing in many ways, especially in a Continuous Integration (CI) environment where ideally all tests must pass before a change can be integrated, or in other words any failing test must be fixed before a release. Test cases that fail non-deterministically jeopardize any good practice of prioritizing those tests that recently failed, because as explained in [28] this would end up in executing mainly flaky tests.

Indeed, in Google, almost 16% of individual tests contain some form of flakiness [29], and these flaky tests are the cause of 84% of all observed transitions (i.e., changes from pass to fail or the vice versa for the test results across project commits) [23]. A non negligible percentage of flaky tests is observed also in Microsoft: while monitoring five projects over a one-month period, 4.6% individual tests were identified as flaky [21]. Open Source (OS) projects do not escape flakiness either: a study of 61 projects using Travis CI assessed that 13% of all observed failures were attributable to flakiness [20]. A similar percentage of 12% flaky tests on average was observed for test cases executed in the IDE over 3,500+ both industrial and OS projects [5].

In addition to wasting developers' effort in debugging a System-under-Test (SUT) that is actually correct because an observed failure is due to a flaky test and not to the latest introduced change, flakiness also inflates testing time: several CI platforms now routinely rerun failing test cases a number of times to ascertain that failures are not intermittent. This in turn produces other indirect costs in either case that a test is eventually marked as flaky or non-flaky: if flaky, after receiving several false alarms, testers tend to lose trust in the process and be less reactive to failures [29]; if non-flaky, the failure was real, but its debugging has been delayed.

As said, in practice flakiness is identified by rerunning a failed test several times, e.g., even up to 10 times [30]. Approaches have been proposed to reduce the implied overhead, e.g., by coupling test rerunning with code analysis [4, 22]. In this way even one execution might be sufficient. In the following we denote flakiness detection techniques that rely on test execution (one or more times) as *dynamic* techniques.

In contrast, some approaches have recently been proposed to recognize flakiness based on characteristics of known flaky tests, e.g., [16, 18, 43]. Such approaches do not require test rerunning, but either rely on manual identification of such characteristics by experts [18, 43], or learn them from a vast historical dataset [16]. In either case they require large effort to be generalized.

As wished by Harman and O'Hearn [14], research should find a quick yet effective method for test flakiness assessment. Indeed, we concur with Gyori and coauthors when they state - talking about state-polluting test cases (i.e., tests that can originate flakiness) - that "*ideally a polluting test should be caught right when the developer is about to add it to the test suite because that is when the developer is in the best position to fix*" it [13]. Accordingly, our aim is a method that can timely predict flakiness *even before a test is executed.*

Toward such aim, we hypothesize that test flakiness can be early detected by just looking at test code. Some recent studies [9, 31] have shown that test code similarity can provide an effective instrument for test suite prioritization and reduction, however to the best of our knowledge no previous work has considered to *leverage test code similarity for identifying flaky tests.* If the hypothesis were confirmed by the data, such an approach would only need a train sample of test cases labeled as flaky and non-flaky (which can be assumed in current industrial practice [30]) and neither expert

*Also with ISTI-CNR, Pisa, 56124, Italy
†Also with Vrije Universiteit Amsterdam, 1081HV, The Netherlands.

consultancy nor test rerunning nor any further characterizations of flaky tests beyond their test code.

Our contribution. This paper contributes with an approach called FLAST that predicts if a test is flaky based on its similarity with known flaky tests at the negligible cost of a fast training step: as we describe in Section 3, we first map the tests in the train set onto some metric space, where we fix the notions of source-code similarity and distance among test cases; then, we train on it a k-Nearest Neighbor classifier [2] to predict the flaky or non-flaky nature of any new test.

Our extensive analysis on 24 projects obtained from repositories used in previous studies [4, 22, 34] supports our hypothesis. FLAST yields Median values for precision varying from 0.87 (with 0.70 recall) up to 0.98 (with 0.18 recall). Different precision values depend on a threshold we can tune to make the approach more or less conservative: lower precision value of 0.87 corresponds to no threshold; higher threshold values increase precision reducing the percentage of false positives down to 0.03%, but at the cost of missing about 40% of potential flaky tests. Indeed, if FLAST predictions are returned to developers for manual fix of flakiness, it would be essential to contain false positives, i.e., claiming a test as flaky when it is not, to the lowest possible degree. Instead, if FLAST predictions are processed automatically higher recall values (even though corresponding to lower precision) may be preferable.

In addition, provided that flaky tests in the train set are categorized according to their cause (e.g., as in the dataset provided by [34]), FLAST can be trained to also detect the specific flakiness categories. We got Median precision varying between 0.82 and 0.98.

Summarizing, we provide the following contributions:

- *Idea:* first method for similarity-based flakiness detection based on static analysis of test code: we can predict both if a test is flaky, and its cause (in case the train set also includes flakiness labels as in [34]).
- *Implementation:* a tool prototype of FLAST that embeds fine-tuned heuristics to enhance precision and efficiency.
- *Evaluation:* a study over 24 projects including in total more than 60K test methods, 3,700+ of which flaky. The whole dataset and all results are made available along with the tool for study replication.

Given its lightweight nature, generality, and high precision, we envisage that FLAST can be embedded within any CI platform with great returns in reducing the incidence of test flakiness and maintaining code velocity.

In the next section we overview related work. Then we present the FLAST approach (Section 3) and the methodology followed to evaluate it (Section 4). We provide study results (along with threats-to-validity) in Section 5. Finally, we discuss application of FLAST in CI practice (Section 6) and draw conclusions (Section 7).

2 RELATED WORK

Until the cited study by Luo et al. [26], the scientific literature almost ignored test flakiness. A few works mentioned the problem as an aside issue, but to our knowledge no research paper centrally focused on flakiness. Instead, in the last five years flaky tests are drawing increasing attention, also triggered by practitioners' alerts about the relevance and spread of the problem [10, 14, 29].

Aiming at a comprehensive review of efforts undertaken so far to address flaky tests, we launched a query over three wide digital libraries, namely ACM DL,[1] IEEExplore,[2] and Scopus.[3] We searched for string *"flaky test" OR "flaky tests" OR flakiness* in the abstract (ACM DL) or in the metadata (IEEExplore and Scopus).

As a result, after removing duplicates and excluding not relevant works, we collected a total of 19 primary studies that are mainly or in large part dealing with flakiness. This result is not to be meant as a systematic survey of the topic, which is beyond the scope of the present section and would deserve a dedicated paper: for instance we did not search for common "flaky" synonyms such as "*intermittent*" or "*non-deterministic*," nor we performed full-cycle snowballing iterations (but in part we did).

For ease of exposition, we can classify such 19 studies into three main groups: *i) empirical studies* on flaky tests [20, 26, 33, 34, 36, 37, 41, 42], *ii)* approaches to *detect flaky tests* [4, 16, 18, 22, 39, 43], and *iii)* approaches to prevent, mitigate, repair or, in a single word, *manage flaky tests* [11, 13, 21, 27, 38]. We briefly overview these categories.

Empirical studies. Luo et al. [26] provide the first extensive study of flakiness by inspecting more than one thousand commits that likely fixed flaky tests over the central repository of the Apache Software Foundation. Their results provide a *must-read* to understand the phenomenon of flaky tests, and is highly cited especially with reference to their ten categories of flakiness root causes ranked by incidence. Over the same repository, Vahabzadeh et al. [42] studied more broadly all possible types of bugs in test code and observed that flaky tests, together with semantic bugs, constitute the dominant cause of tests provoking false alarms. Thorve et al. [41] conducted a study of flaky tests in Android apps. They observed that some causes of Android tests flakiness are similar to those identified in [26], but two new causes are Program Logic and UI. Presler-Marshall et al. [36] report on a specific study on flakiness in web testing using the Selenium tool. They provide several hints on the effect of different test environment configurations. The study by Labuschagne et al. [20] of 61 projects from GitHub Archive focuses on the costs of maintaining a regression test suite; they observe that ~13% of test failures are due to flaky tests. Rahman and Rigby [37] studied the crash reports submitted by Firefox users in Beta and production stages, asking among other things how many of them are associated with flaky tests. They observed that developers are more conservative with known flaky tests when releasing production builds than Beta builds. In a series of two studies [33, 34], Palomba and Zaidman analyze the relation between flakiness and *test smells*, i.e., suboptimal choices in test development. Notably they observed that 75% of the flaky tests were due to presence of smells; they also drew a different ranking of possible flakiness causes than that in [26].

Detecting flaky tests. In practice flaky tests are detected by rerunning either all failed test cases or the suspect ones (e.g., the tests that transited from pass to fail) a number of times, e.g., 10 times [30]. Following [4], we call this generic strategy as Rerun.

[1] https://dl.acm.org
[2] https://ieeexplore.ieee.org
[3] https://www.scopus.com. Note that in Scopus the search was limited to the Computer Science subject area.

In the literature we found few works proposing approaches that improve on Rerun, which is costly and not very precise. Among these, the authors of [4, 22, 39] propose *dynamic* techniques, i.e., they still base the identification on test case execution (one or more times), but enhance precision with deeper analyses. In contrast, the works in [16, 18, 43] leverage knowledge of flakiness to build a *static* predictor. In [43], a set of problematic test code patterns (actually not exclusively for flakiness, but broadly for test code bugs) is manually elicited. The work in [16] is similar, but uses a machine learning approach to mine association rules among individual test steps in tens of millions of false test alarms. Finally, in [18] a Bayesian network is constructed.

Managing flaky tests. Some authors have proposed specific techniques to tackle known root causes of test flakiness. One prevalent problem is the existence of dependencies among tests. To prevent it, Gyori et al. [13] propose the POLDET technique that timely detects if a new added test case "pollutes" the state of heap shared by more tests. Instead Gambi et al. [11] develop the PRADET approach, which discovers test dependencies by flow analysis and iterative testing of possible dependencies. Lam et al. [21] detect the root causes of flaky tests by executing them after instrumentation and comparing the logs of passing and failing runs. A different thread of research can be noted in [27, 38]: as prospected in [14], test flakiness is such a pervasive and complex problem that any test technique should include appropriate means to manage it. These two works go in this direction, and propose measures to mitigate the impact of flaky tests on test mutation [38], and on learning-based test selection [27].

This work. We leverage results from the empirical studies, in particular the dataset provided by [34]. Our aim is detecting flaky tests as in [4, 16, 18, 22, 39, 43]. However, FLAST differs from all of them, as to identify a flaky test it does not need neither to execute the test case nor any expert knowledge or extensive domain data. We make a more detailed comparison in Section 5, after having explained how FLAST works. In this paper we do not focus on managing the detected flaky tests, but our approach could be used to provide developers with information about how similar flaky tests have been repaired in the past.

3 APPROACH

Let T be a test suite of which we know the flaky nature, i.e., we know whether each test $t \in T$ is *flaky* or *non-flaky*. More formally, let $\ell : T \to \{0, 1\}$ be the function such that $\ell(t) = 1$ if t is flaky and $\ell(t) = 0$ otherwise, for every $t \in T$. Given an unknown test $s \notin T$, i.e., a test of which we do not know the nature, the idea on which our approach is based is that if s is "similar" (for some notion of similarity) to a test $t \in T$ such that $\ell(t) = 1$, then there is a good chance for s to be flaky as well, because they could share the traits that make both their behaviors non-deterministic: for example, they could be testing the same functionality of the SUT or be both dependent on other test cases or be accessing a same shared resource. In the same way, if s is similar to a test t such that $\ell(t) = 0$, then s has a good chance to be non-flaky.

To actualize such idea we need to find a notion of similarity that can capture the flaky nature of a test: as anticipated in the Introduction, we model the tests in T as points in some metric space, where we fix a notion of source-code similarity and distance among test cases, and then train on T a k-Nearest Neighbor classifier [2]. We provide a visual representation of how FLAST works in Figure 1. The full black symbols are the tests in T, represented as points in a plane; the white symbols, instead, are the tests not in T of which we predict the nature. We look at the neighborhood of each of these tests, i.e., at the tests that are similar according to our representation, and predict if each of them is flaky or not according to the nature of the similar neighbors.

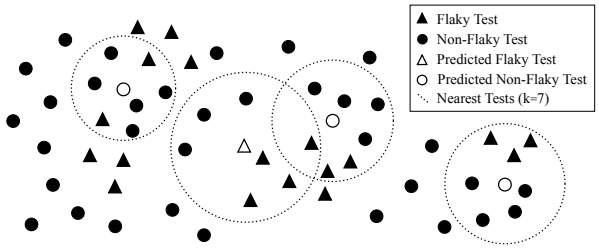

Figure 1: Visual representation of FLAST prediction

Vector space modeling. Similarly to what has been done in [9], we model the tests in T as points in an n-dimensional vector space using the *bag-of-words model* [24]: each test case t is represented as the multiset (i.e., a set that allows multiple instances of its elements) of the tokens composing its source code, split by whitespace characters. According to this model, the dimensionality n of the space induced by T is equal to the number of distinct tokens in the source code of T. Each test $t \in T$ is then represented as a vector $\boldsymbol{t} \in \mathbb{R}^n$ with component relative to token i weighted proportionally to the multiplicity of i among the tokens of t.

Similarity and distance. Given two vectors $\boldsymbol{s}, \boldsymbol{t} \in \mathbb{R}^n$, we measure their similarity using the cosine of the angle θ between them, i.e., via the *cosine similarity* $S_c(\boldsymbol{s}, \boldsymbol{t}) = \cos\theta$, whereby: $\cos\theta = \frac{\langle \boldsymbol{s}, \boldsymbol{t} \rangle}{\|\boldsymbol{s}\| \cdot \|\boldsymbol{t}\|}$, $\langle \boldsymbol{s}, \boldsymbol{t} \rangle = \sum_{i=1}^{n} \boldsymbol{s}_i \boldsymbol{t}_i$ is the dot product between $\boldsymbol{s}$ and $\boldsymbol{t}$, and $\|\boldsymbol{s}\| = \sqrt{\sum_{i=1}^{n} \boldsymbol{s}_i^2}$ is the Euclidean norm of $\boldsymbol{s}$. Instead, we measure their distance via the *cosine distance* $D_c(\boldsymbol{s}, \boldsymbol{t}) = 1 - S_c(\boldsymbol{s}, \boldsymbol{t})$.

Dimensionality reduction. To mitigate the effects of the curse of dimensionality in the neighbor search [7] and obtain gains in terms of efficiency and storage overhead we apply a dimensionality reduction technique called *sparse random projection* [1, 25]. Roughly speaking, points are projected onto a random d-dimensional subspace of $\mathbb{R}^n$, with $d \ll n$, such that the pairwise distance of the projected points is preserved up to a multiplicative factor ε [17]. Herein, we set $\varepsilon = 0.33$, but it can be customized to have a different effectiveness/efficiency tradeoff in the distance measurement. The dimensionality d of the random subspace onto which points are projected is independent from the initial dimensionality n, i.e, from the content of the tests, but only depends on the number of tests in T and its much smaller that them, being $d \in \Omega\left(\frac{\log |T|}{\varepsilon^2}\right)$.

Flakiness prediction. After modeling the tests in T as vectors and reducing their dimensionality, we predict the nature of an unknown test case $s \notin T$. In particular, as previously mentioned,

Algorithm 1 FLAST Prediction

Input: Test suite T; Function ℓ; Test s; #Neighbors k; Threshold σ
Output: Flakiness prediction for s

1: $N_s \leftarrow \underset{R \subseteq T \,:\, |R|=k}{\arg\min} \sum_{t \in R} D_c(\boldsymbol{s}, \boldsymbol{t})$ ▷ k nearest neighbors
2: $\phi_s \leftarrow \sum_{t \in N_s \,:\, \ell(s)=1} \frac{1}{D_c(\boldsymbol{s}, \boldsymbol{t})}$ ▷ *flakiness* measure
3: $\psi_s \leftarrow \sum_{t \in N_s \,:\, \ell(s)=0} \frac{1}{D_c(\boldsymbol{s}, \boldsymbol{t})}$ ▷ *non-flakiness* measure
4: **if** $\frac{\phi_s}{\phi_s+\psi_s} > \sigma$: **return** True ▷ predict the test is *flaky*
5: **else**: **return** False ▷ predict the test is *non-flaky*

we use a k-Nearest Neighbors classifier and train it on the vector representation of the tests in T. The value of k sets the tradeoff between variance and bias in the classification: a low value of k makes the classification more subject to noise (increased variance), while a high value of k smooths the decision boundaries (increased bias). As a general rule of thumb one would set $k = \sqrt{|T|}$, but in this paper we set $k = 7$ (without optimizing it through cross validation) given the unbalance in the datasets used in the experiments (some projects has less than 7 flaky tests in the train set). The flakiness prediction performed by FLAST is sketched in Algorithm 1.

First, the unknown test case s in mapped to a vector $\boldsymbol{s}$ in the same vector space used for the tests in T.[4] Then, FLAST searches for the set N_s of k tests that are closer to $\boldsymbol{t}$ according to the cosine distance of their vector representations (Line 1); we look for the neighbors via a naive linear search, but the same operation could be done using other techniques. The *flakiness* and *non-flakiness* measures of s, i.e., ϕ_s and ψ_s (Lines 2-3), are computed as a function of the neighborhood of s, weighting the nature of each neighbor by the inverse of its cosine distance to $\boldsymbol{s}$. Test s is predicted to be *flaky* if $\frac{\phi_s}{\phi_s+\psi_s} > \sigma$, for some threshold $\sigma \in [0, 1]$, and to be *non-flaky* otherwise (Lines 4-5); using ϕ_s and ψ_s we emphasize the similarity between s and its flaky/non-flaky neighbors, rather than their sole number; a threshold parameter $\sigma > 0.5$ makes the algorithm more conservative in predicting a test as flaky.

Flakiness category identification. There may exist different root causes of flaky tests, such as *concurrency* and *test order dependency*, that allow for a classification into multiple flakiness categories, as done, e.g., in [26, 34]. In the scenario that for each flaky test $t \in T$ we also know its *flakiness category*, in addition to predicting the nature of an unknown test case $s \notin T$, we can also identify its flakiness category via a majority voting strategy. Let C_s be the multiset of possible flakiness categories of s, computed as the union (with multiplicities) of all flakiness categories of its k nearest neighbors: we predict the category of flakiness of s as the most frequent flakiness category in C_s.

[4] Potential new tokens of s w.r.t. those of T do not affect the distance computation after the random projection. The new tokens (say they are m) would change the random projection matrix and the vectors in T should be re-projected from the "augmented" $(n + m)$-dimensional space; however, the new matrix would map the vectors in T onto the same space they are currently projected on, since the components in the augmented space relative to the new tokens of s would be 0.

4 EVALUATION

We aim at evaluating the effectiveness and efficiency of FLAST as an approach for predicting test flakiness. In this section we describe the research questions, the methodology we followed to answer them, and the experimental setting.

4.1 Research questions

The first obvious question is whether FLAST is actually able to detect flaky tests, so we ask:

RQ1: How *effective* is FLAST in predicting test flakiness?

As prediction relies on similarity, we also inquire how large a train sample of known flaky and non-flaky tests would FLAST need, i.e.:

RQ2: How does FLAST effectiveness *vary with the size* of the train sample?

If the known flaky tests are classified into categories, can we also use FLAST to predict the cause of flakiness, or:

RQ3: How effective is FLAST in identifying a *flaky test category*?

For practical adoption we need also to evaluate the costs of FLAST, but evaluating actual costs of putting it in production is a very complex task. As a first step we ask:

RQ4: How *efficient* is FLAST in terms of *training time*, *prediction time*, and *storage overhead*?

Finally, it is also important to evaluate FLAST in comparison with other existing approaches, so we also investigate:

RQ5: How does FLAST *compare with other* state-of-the-art techniques?

4.2 Evaluation methodology

To answer RQ1. We measure Precision P and Recall R:

$$P = \frac{TP}{TP + FP}, \qquad R = \frac{TP}{TP + FN},$$

where TP, FP, and FN respectively denote *true positive*, *false positive*, and *false negative* predictions. We will also derive Confusion Matrices, which allow us to look in detail at the absolute numbers and percentages of FPs (which for a flakiness predictor we would like to keep limited). We infer such metrics through Stratified 10-fold Cross Validation, a standard procedure for validating ML methods [19]. The dataset is split into 10 folds, each used once as the test set while the remaining 9 folds are used as the train set. Stratification, instead, ensures that each fold is a good representative of the original dataset by preserving the proportion of flaky tests and reducing both bias and variance of the classifier [19].

To answer RQ2. We use the same metrics defined for RQ1, but considering different sizes of the train set, ranging from 10% to 90% (with a step size of 5%) of the size of the dataset. Differently from RQ1, here we use Stratified Shuffle Split with 10 splits, an alternative to k-fold Cross Validation that allows a finer control on the train/test split. In fact, this validation strategy allows us to specify the train set size.

To answer RQ3. Whenever we classify a test as flaky we also identify its *flakiness category*.[5] To evaluate effectiveness we count the fraction of tests predicted as flaky and for which the predicted

[5] We assume that flaky tests are labeled with their categories, as for instance in [34].

category is among the real ones. More formally, let S be a set of tests predicted as *flaky* by FLAST. For each test $t \in S$, let F_t be the set of real flaky categories of t and let f_t be the flaky category of t predicted by FLAST. Let F be the overall set of possible flaky categories, i.e., $F = \bigcup_{t \in S} F_t$. We measure the effectiveness of FLAST in identifying a flaky test category as

$$A_{FC} = \sum_{t \in S} \frac{\mathbf{1}_{F_t}(f_t)}{|S|},$$

where $\mathbf{1}_{F_t} : F \to \{0, 1\}$ is the indicator function of the flaky category, i.e., $\mathbf{1}_{F_t}(f_t) = 1$ if $f_t \in F_t$ and $\mathbf{1}_{F_t}(f_t) = 0$ otherwise.

Moreover, we also compute FLAST precision in predicting each individual flaky category found in the ground truth data, i.e., the fraction of samples for which FLAST correctly guessed a category among those that FLAST predicted in that category.

To answer RQ4. We measure the *training time* (time to vectorize the tests in the train set and to build the data structure containing them), the *prediction time* (time to vectorize a new test, query the data structure for the k nearest neighbors, and predict its nature) and *storage overhead* (of the "trained model," i.e., of the data structure to be stored on disk after the training phase).

To answer RQ5. As described in Section 2, our search of literature identified seven approaches [4, 16, 18, 22, 30, 39, 43] as competing approaches. Performing an empirical comparison of FLAST against those entails a cumbersome and time consuming process, which would not lead to meaningful results, due to the drastic different nature of the approaches, especially if we consider: *i)* their scopes (FLAST is applied to every test case, whereas some of the approaches are applied to only a test subset, e.g., those that failed, and some can only identify specific types of flakiness), and *ii)* the required inputs and utilized resources (e.g., FLAST exclusively relies on test source code, while other approaches also require additional information, such as manual input provided by experts [18, 43]). Hence, to answer RQ5, we leverage a qualitative methodology, carried out by eliciting a set of prominent characteristics of flaky test detection approaches.

4.3 Experimental setting

Evaluation dataset. In order to answer RQ1…RQ4, we leverage an experimental dataset encompassing over 3.7K real-life flaky tests belonging to 24 distinct software projects. This dataset is obtained by combining flaky tests datasets that are publicly available [4, 22, 34]. The integration of different existing and already used datasets allows us to efficiently gather a large amount of data for experimentation, while ensuring data heterogeneity and high-quality. From the original datasets, we select those projects that contain enough data for training FLAST: precisely we include in our dataset all the projects that contain at least 7 flaky tests.[6]

Replication package. The entirety of the software projects included in our dataset is hosted on GitHub.[7] In the original dataset each flaky method is mapped to a unique commit hash,[8] a classpath, and a method name. We use this information by performing the following steps: *i)* checking out the version of the software projects where flaky tests are present, *ii)* identifying the flaky methods via their given classpath and method name, *iii)* parsing the flaky methods and storing each one in a separate file for subsequent analysis, and *iv)* parsing and storing the remaining methods of the classes present in the test suites (which by exclusion are assumed as non-flaky). An overview of the data selected from each original dataset is provided in Table 1.

Table 1: Overview of the evaluation datasets

Source	#Projects	#Flaky Methods (SLOC)	#Total Methods (SLOC)
Palomba et al. [34]	11	3,424 (72,865)	33,740 (717,655)
iDFlakies [22]	9	258 (4,824)	3,705 (52,023)
DeFlaker [4]	4	57 (891)	3,429 (36,079)
Total	24	3,739 (78,580)	40,874 (805,757)

Threshold σ. As introduced, FLAST can be tuned to work in less or more conservative way by setting a lower or higher value for the threshold $\sigma \in [0, 1]$. We run all experiments under two scenarios: $\sigma = 0.5$ and $\sigma = 0.95$. With reference to Algorithm 1, the former scenario corresponds to using no threshold, i.e., we predict a test s is flaky if $\phi_s > \psi_s$; the latter puts a high threshold so that a test s is predicted as flaky only when we are highly confident. We expect that the second scenario will predict less flaky tests, but with higher precision than the first one (and hence less false positives). Contrariwise, for the $\sigma = 0.5$ scenario we expect higher recall.

Hardware. All experiments were run on a MacBook Pro with a 2.7 GHz Intel Core i5, 8 GB RAM, running macOS Mojave 10.14.6.

Qualitative comparison. To answer RQ5, we elicited a set of features that we consider relevant in choosing an approach for flaky test prediction: these features were derived by consensus among the authors, and are are quickly described below:

- *Analysis type*: Possible values are Static (no test execution needed) or Dynamic (test must be executed at least once).
- *SUT coverage*: Possible values are YES (approach uses SUT coverage reports) or NO otherwise.
- *Flakiness type*: Possible values are Generic (approach targets any type of flaky test) or Specific (only some specific types of flakiness can be detected).
- *Scope*: Possible values are All (approach is applied to all tests) or Subset (only a part of tests is analyzed).
- *Action type*: Possible values are Proactive (approach actively searches for flaky tests) or Reactive (approach is invoked only in reaction to transitions).
- *Expert knowledge*: Possible values are YES (approach needs expert consultancy) or NO otherwise.
- *Train set*: Possible values are YES (approach needs to be trained on a set of known flaky tests) or NO otherwise.
- *Precision*: Provide where available (or n.a. otherwise) the precision results obtained by the approach authors.
- *Overhead*: Provide where available (or n.a. otherwise) the overhead estimations claimed by the approach authors.

[6] Due to space constraints, the list of all projects considered, as well as the comprehensive list of flaky classes, flaky methods, commit hashes, and the entirety of the source code utilized as data in our experiments, is made available in our replication package.
[7] https://github.com/ICSE2020-FLAST/FLAST
[8] As FLAST does not require historical data, if a flaky method is associated to more than one commit hash in the original dataset, only the most recent hash is considered.

With regard to precision and overhead, we warn that the reported values may not be comparable among each other, as they were obtained under different experimental conditions. These features constitute the columns of Table 3 discussed in Section 5.5.

5 RESULTS

We report below the results obtained to answer the five RQs and then discuss potential threats to validity.

5.1 [RQ1] How *effective* is FLAST in predicting test flakiness?

The box plots of Figure 2 show the distribution of precision and recall values obtained by FLAST when applied over the evaluation datasets. For these metrics, the higher the result (reported in the vertical axis), the better. The performance results are presented on the union of all datasets ("All"), and also grouped by dataset. For each metric displayed (Precision or Recall), the left (blue) box refers to the results for the scenario with threshold $\sigma = 0.5$, while the right (orange) box refers to the scenario with threshold $\sigma = 0.95$. A detailed breakdown of precision and recall values per threshold and per project is available in Table 2. The number of flaky methods and total number of methods per project is also available from the same table, Columns 2 and 3, respectively.

When considering the consolidated results ("All") for the less conservative scenario ($\sigma = 0.5$), an overall average precision of ~0.83 was obtained (i.e., when FLAST classified a test as flaky, it got it right ~83% of the times). When we increase the threshold to $\sigma = 0.95$, the overall average precision increases to ~0.92. Similar results were observed when considering the evaluated datasets individually, with the exception of the DeFlaker dataset for which the precision actually decreased when FLAST became more conservative (this unexpected result could be due to the very small number of flaky tests in this dataset).

The confusion matrices displayed in Figure 3 provide a detailed analysis of the performance of FLAST for each class (flaky or non-flaky) individually. They show us the ways in which FLAST is "confused" when it makes predictions. This is an important aspect to be taken into account, specially for imbalanced problems like ours. A flaky test correctly classified as flaky by FLAST is counted as a true positive (top left), whereas if it is classified as non-flaky it is counted as a false negative (bottom left). Analogously, if a non-flaky test is correctly classified as non-flaky it is counted as a true negative (bottom right), whereas it counts as a false positive (top right) if it is incorrectly classified as flaky. False positives (*FPs*) and false negatives (*FNs*) are sometimes referred to as Type I and Type II Errors [35]. Committing a Type I Error is critical if tests predicted as flaky are sent back to developers for manual analysis and repair, as this could waste developer's time looking for flakiness in a test that is actually non-flaky [16]. Ideally, in such a case we would like to have maximum precision even if this comes at the expense of a diminished recall. Conversely, we could envisage a different scenario in which FLAST results are sent to a tool, e.g., for test prioritization or for performing a dynamic analysis. In this case we might prefer to trade-off between Precision and Recall, containing also Type II Errors.

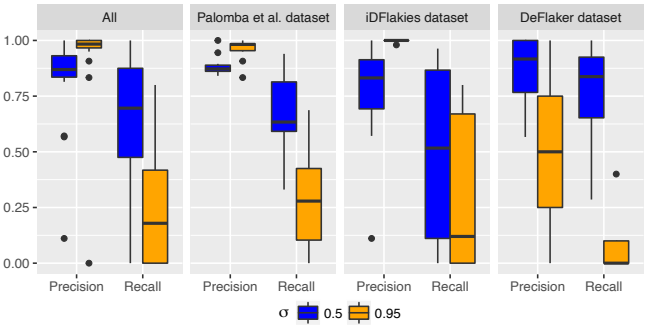

Figure 2: FLAST effectiveness over the evaluation datasets

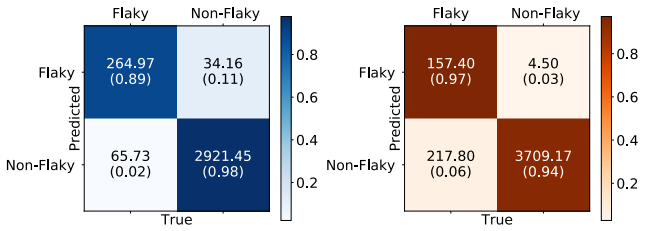

Figure 3: Confusion matrices from experimental results of FLAST, with absolute and normalized values in each cell (on the left (blue) $\sigma = 0.5$; on the right (orange) $\sigma = 0.95$)

From Figure 3 we can see clearly that by increasing the threshold from $\sigma = 0.5$ to $\sigma = 0.95$, the percentage of TPs[9] is improved from 0.89 to 0.97, implying that the percentage of FPs goes from 0.11 down to 0.03. Looking at the absolute numbers we can notice that FLAST is more conservative in classifying a test as flaky. An improved precision comes at the cost of missing many more flaky tests (reduced recall); such missed flaky test can still be detected using a traditional rerunning strategy.

5.2 [RQ2] How does FLAST effectiveness *vary with the size* of the train sample?

Figure 4 shows the precision and recall values achieved by FLAST for different training sample sizes. The x-axis displays the proportion of samples used for training, while the y-axis reports the score achieved by each metric. The lines for the threshold $\sigma = 0.5$ are in blue and those for the threshold $\sigma = 0.95$ are in orange. Precision is represented by the solid lines, while recall is illustrated by the dashed lines.

One common trend observed for both thresholds is that precision seems to be almost stable across the different training sample sizes, while recall tends to improve as we increase the amount of samples used for training. FLAST with $\sigma = 0.95$ is more precise than its less conservative version for all the sizes of training sample considered. On the other hand, its recall is much lower than that of FLAST with $\sigma = 0.5$. That is an expected behavior due to the unavoidable

[9]It may be worth noticing that this percentage is calculated by cumulating TPs and FPs across all projects and would correspond to a "cumulative" Precision value, which differs from the Precision values of Table 2, obtained as the Mean and Median of Precision values of the projects.

Table 2: Experimental measurements of FLAST effectiveness and efficiency

Dataset [Source]	#Flaky Methods	#Total Methods	Precision Threshold $\sigma = 0.5$	Recall Threshold $\sigma = 0.5$	A_{FC} Threshold $\sigma = 0.5$	Precision Threshold $\sigma = 0.95$	Recall Threshold $\sigma = 0.95$	A_{FC} Threshold $\sigma = 0.95$	Train Time (s)	Predict Time (ms)	Storage (KB)
activiti [22]	20	53	0.88	0.95	0.92	1.00	0.80	1.00	0.06	1.55	181.58
ant-ivy [34]	764	1,175	0.94	0.93	0.91	0.98	0.69	0.98	0.36	0.46	4,049.44
apache-derby [34]	84	10,144	0.86	0.33	0.71	1.00	0.01	0.67	5.26	1.10	19,717.79
apache-hadoop [22]	68	1,121	0.94	0.52	0.69	1.00	0.17	1.00	0.59	0.72	3,306.22
apache-hbase [34]	277	2,924	0.86	0.61	0.81	0.98	0.28	0.95	1.73	0.87	7,476.37
apache-hibernate-orm [34]	127	3,830	0.86	0.77	0.95	0.98	0.41	1.00	1.41	0.67	5,021.31
apache-hive [34]	106	647	0.88	0.63	0.87	0.95	0.23	0.90	0.39	0.77	2,047.71
apache-incubator-dubbo [22]	21	507	0.57	0.35	0.63	-	0.00	-	0.25	0.66	1,450.07
apache-karaf [34]	120	385	0.87	0.76	0.86	1.00	0.34	1.00	0.22	0.70	1,216.72
apache-nutch [34]	184	257	0.87	0.94	0.74	0.91	0.44	0.92	0.16	0.80	719.26
apache-pig [34]	1,268	4,796	0.88	0.86	0.84	0.97	0.49	0.94	2.32	0.83	14,551.60
apache-qpid [34]	271	2,166	0.84	0.61	0.78	0.99	0.18	0.98	0.92	0.64	7,031.94
apache-wicket [34]	216	2,082	0.89	0.52	0.80	0.83	0.02	0.00	0.86	0.60	3,806.13
elastic-job-lite [22]	10	785	-	0.00	-	-	0.00	-	0.33	0.61	2,095.35
handlebars [4]	7	530	1.00	1.00	-	-	0.00	-	0.27	0.67	1,405.26
http-request [22]	28	168	1.00	0.87	1.00	1.00	0.73	1.00	0.11	0.84	756.45
java-websocket [22]	52	488	0.81	0.96	0.80	0.98	0.67	1.00	0.23	0.62	1,598.06
logback [4]	11	1,052	0.57	0.90	-	1.00	0.40	-	0.51	0.67	2,298.47
lucene-solr [34]	7	5,334	1.00	0.57	1.00	-	0.00	-	3.04	0.97	10,084.87
okhttp [4]	32	1,231	0.83	0.78	-	0.00	0.00	-	0.67	0.76	3,570.17
retrofit [22]	9	424	0.11	0.11	0.08	-	0.00	-	0.17	0.53	1,875.80
tachyon [4]	7	616	1.00	0.29	-	-	0.00	-	0.34	0.71	2,052.89
vertx-rabbitmq-client [22]	7	44	-	0.00	-	-	0.00	-	0.06	1.80	145.58
wildfly [22]	43	115	0.83	0.78	0.84	1.00	0.12	1.00	0.08	0.95	409.41
Mean [34]	311.27	3,067.27	0.89	0.69	0.84	0.96	0.28	0.83	1.51	0.76	6,883.92
Median [34]	184.00	2,166.00	0.87	0.63	0.84	0.98	0.28	0.95	0.92	0.77	5,021.31
Mean [22]	28.67	411.67	0.74	0.50	0.71	1.00	0.28	1.00	0.21	0.92	1,313.17
Median [22]	21.00	424.00	0.83	0.52	0.80	1.00	0.12	1.00	0.17	0.72	1,450.07
Mean [4]	14.25	857.25	0.85	0.74	-	0.50	0.10	-	0.45	0.70	2,331.70
Median [4]	9.00	834.00	0.92	0.84	-	0.50	0.00	-	0.43	0.69	2,175.68
Mean	155.79	1,703.08	0.83	0.63	0.79	0.92	0.25	0.89	0.85	0.81	4,036.19
Median	47.50	716.00	0.87	0.70	0.82	0.98	0.18	0.98	0.35	0.71	2,074.12

The value σ is the threshold used by FLAST (see Algorithm 1, Line 4). The values reported in the table are mean values of the 10-Fold Cross Validation. Precision and A_{FC} values equal to "-" mean that the metric is undefined, e.g., FLAST did not predict any test as flaky or ground truth data for flaky category identification were not available.

tradeoff between precision and recall: improving one metric tends to be associated with poorer performance of the other [8].

While Figure 4 is important to give us some hints that the precision of FLAST is not strongly affected by training sample size (at least not as much as recall), to get an intuition about the number of already-labeled test cases required to make FLAST work with high precision rates (above 90%), we need to look closely at the data. We observe that for 5 subjects[10] FLAST ($\sigma = 0.5$) is able to train the model with less than 100 test cases (average = 47) and achieves an average precision of ~0.98 while predicting the nature of 2,438 tests. Similar results are also observed when adopting $\sigma = 0.95$, but for 6 subjects[11] instead of 5. Note that the set of subjects for which FLAST ($\sigma = 0.95$) has high precision is almost a superset of the other, i.e., if FLAST ($\sigma = 0.5$) had high precision on the subject then FLAST ($\sigma = 0.95$) has either high precision or remains conservative (does not predict any test as flaky). Looking from a different perspective, for 3 subjects[12] FLAST ($\sigma = 0.95$) achieved precision rates above 90% with 10 or less test cases labeled as flaky in the training set.

[10]The 5 subjects are: activiti [22], elastic-job-lite [22], http-request [22], logback [4], and tachyon [4].
[11]The 6 subjects are: activiti [22], ant-ivy [34], apache-karaf [34], apache-nutch [34], http-request [22], and wildfly [22].
[12]The 3 subjects are: activiti [22], http-request [22], and logback [4].

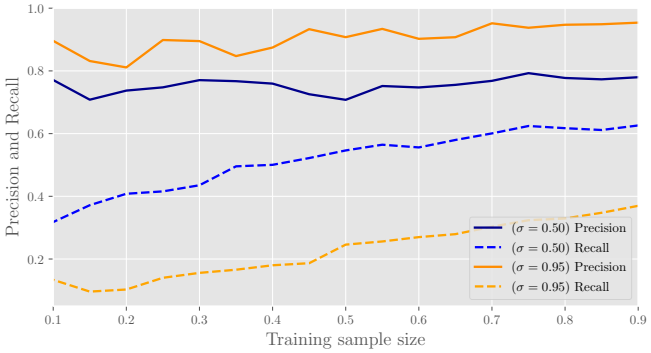

Figure 4: FLAST effectiveness for different train set sizes

5.3 [RQ3] How effective is FLAST in identifying a *flaky test category*?

In Table 2 the results of FLAST relative to A_{FC} are reported. FLAST ($\sigma = 0.5$) achieves Mean and Median A_{FC} values respectively of 0.79 and 0.82, meaning that whenever FLAST classified a test as flaky it also correctly guessed at least one of its flaky categories more roughly 80% of the times. Similarly to what happened with precision values discussed for RQ1, FLAST ($\sigma = 0.95$) yields higher

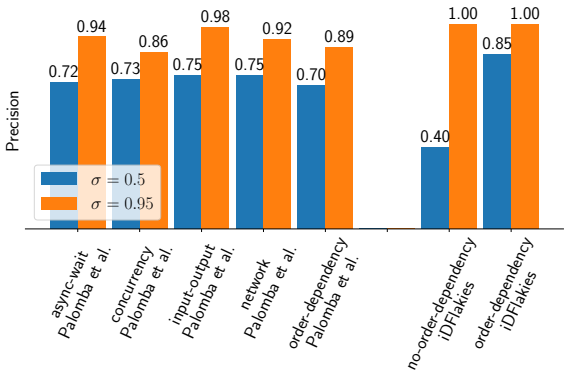

Figure 5: FLAST average precision in the identification of flakiness categories

Mean and Median A_{FC} values, respectively achieving 0.89 and 0.98. The higher value of A_{FC} is strictly correlated with the more conservative behavior obtained setting a higher threshold. Moreover, this gives insights on the fact that the higher is the similarity of a test with a flaky one the higher is the probability that at least one of the flaky categories of the two tests matches.

The bar plots in Figure 5 display how precise is FLAST in the identification of the cause of flakiness for an unknown test case predicted as flaky. We could do this only for the projects in the Palomba et al. dataset [34] and in iDFlakies dataset [22]. The results are grouped by evaluated dataset, then by flaky category, and they represent the average precision obtained across all the projects available for that group. For each group, the left (blue) bar displays the results obtained by FLAST when the threshold is set to $\sigma = 0.5$, while the right (orange) bar shows the results with $\sigma = 0.95$.

Looking at the leftmost group (Palomba et al. dataset [34]), overall FLAST had a good performance regardless of the threshold set: it was able to correctly predict at least one of the real causes of flakiness, of an unknown test case, in at least 70% of the time. With threshold $\sigma = 0.95$, the prediction ability of FLAST was improved for all the categories with an average relative improvement of ~26%. More noticeably, the relative performance of FLAST ($\sigma = 0.95$) improved by ~31% for the *input-output* category.

The difference between the two thresholds was more pronounced for the rightmost group (iDFlakies dataset [22]): while FLAST assigned the correct flaky cause for all the cases when configured to be more conservative ($\sigma = 0.95$), its prediction ability varied greatly with $\sigma = 0.5$.

5.4 [RQ4] How *efficient* is FLAST in terms of *training time, prediction time,* and *storage overhead*?

The experiments were run using a naive brute force algorithm to find the k nearest neighbors. This approach looks for the k nearest neighbors in a linear fashion and has a cost of $O(|T|)$ (considering k constant, as in our experiments) to predict the nature of a test $s \notin T$. As can be observed in Table 2, this strategy can be considered efficient for the projects considered in the experiments, resulting in an average train time and prediction time respectively under one second and one millisecond, and an average storage overhead of roughly 4 MB (with an average project size of ~1,700 test methods).

The Nearest Neighbor problem, in general, can be also approached in other ways, e.g., with the use of space partitioning data structures such as `kd-tree` [6] and `balltree` [32] or in an approximate way using techniques such as Locality Sensitive Hashing [24], that could be of interest to further speed up the computation of the nearest neighbor search. Space-partitioning data structures, though, are not suitable for efficiently finding the nearest neighbors in very high dimensional spaces, as the one induced by a bag-of-word representation of test source code. To use such data structures we should have the dimensionality $d \ll |T|$, which is unlikely also with test suites consisting of millions of methods unless drastically reducing the dimensionality of the space at the cost of weakening the distance approximation guarantees of the JohnsonâĂŞLindenstrauss Lemma [17]. Approximate nearest neighbor search, instead, would allow for a sublinear search time, but getting approximate results which are not ideal in our setting where effectiveness is more important than efficiency, and efficiency is not really a issue.

5.5 [RQ5] How does FLAST *compare with other* state-of-the-art techniques?

In Table 3 we show our classification of the seven identified competing approaches (listed in *1st column*) along the dimensions introduced in Section 4.3; the last row classifies FLAST. Four approaches are dynamic and three are static as is ours (*2nd column*). Only one approach, viz. DeFlaker, relies on coverage reports (*3rd column*): indeed, collecting coverage may be quite costly in CI practice [15] and because of this DeFlaker itself proposes a lightweight technique. Two of the approaches, viz. NonDex and Pattern search, can only detect some specific type of flaky tests (*4th column*), because as reported in the table notes (d) and (e), they rely on pre-determined causes of flakiness. Concerning the approach based on learning Association Rules, this actually targets false test alarms which are a superset of flakiness. Not all approaches are applied to every test case, as we do (*5th column*); Rerun, DeFlaker, iDFlakies, and Association rules analyze test cases based on their outcome, thus they can lose valuable time before detecting flakiness and also could possibly miss flaky tests if they do not fail or pass as expected in the observation window. Almost all approaches, but Rerun and Association rules, take action in proactive way for detecting flaky tests (*6th column*). A critical features is whether an approach is fully automated, or otherwise it requires manual effort for customization/preparation. The latter is the case for three approaches, viz. NonDex, Pattern-search, and Bayesian Network (*7th column*), and clearly may heavily affect their practical adoption. In contrast FLAST, as Rerun, DeFlaker, iDFlakies, and Association Rules, does not require any human consultancy. More than half of the approaches requires a training phase, as does FLAST (*8th column*).

Based on the above classification, the approach most similar to FLAST is the one leveraging Bayesian networks, which shares with ours six features (static, no coverage, generic, whole scope, proactive, and training set), but it differs for an important aspect that is expert knowledge (needed to build the Bayesian network).

Concerning precision and overhead (*9th and 10th columns*), we recall that in Table 3 we report the results obtained by the authors

Table 3: Qualitative comparison among existing flakiness detection techniques

Approach	Analysis type	SUT coverage	Flakiness type	Scope	Action type	Expert knowledge	Train set	Precision	Overhead (time)
Rerun [29]	Dynamic	No	Generic	Subset	Reactive	No	No	n.a.	600+% [3]
DeFlaker [4]	Dynamic	Yes[(a)]	Generic	Subset	Proactive	No	No	0.94[(b)]	4.6%
NonDex [39]	Dynamic	No	Specific[(c)]	All	Proactive	Yes	Yes	n.a.	n.a.
iDFlakies [22]	Dynamic	No	Generic	Subset	Proactive	No	No	n.a.	*Pre-set*
Pattern search [43]	Static	No	Specific[(d)]	All	Proactive	Yes	Yes	1[(e)]	~90 sec
Association Rules [16]	Static	No	Generic[(f)]	Subset	Reactive	No	Yes	0.85÷0.90	n.a.
Bayesian Network [18]	Static	No	Generic	All	Proactive	Yes	Yes	n.a.	n.a.
FLAST	Static	No	Generic	All	Proactive	No	Yes	0.83÷0.92[(g)]	< 1 sec[(h)]

Notes: (a) differential coverage; (b) precision computed over *new failures*; (c) flakiness due to ADINS (Assumes a Deterministic Implementation of a Non-deterministic Specification) code; (d) timing dependency (one among several test code faults targeted); (e) precision over a random sample manually analyzed; (f) look for false test alarms, which are a superset of flaky tests; (g) Mean values with $\sigma = 0.5$ and $\sigma = 0.95$; (h) time for training over a set of ~1,700 tests.

in their experiments, which could not be comparable one with another. Actually, several of the referred studies did not provide values of precision or overhead (the corresponding cells are labeled as *n.a.*). In some works varying precision values are observed and we report their range as min ÷ max. Concerning overhead time, please note that the column shows different units (% of test suite execution time vs seconds) across rows: this reflects faithfully the results reported by authors of referred works. Going in detail, Rerun is not actually a strategy, as various rerunning configurations could be adopted. According to studies in [3, 4], this basic approach is very costly and predicts varying numbers of flaky tests, depending on the conditions under which a test is rerun. For DeFlaker, the authors do not give precision but report TPs and FPs from which we computed an average precision of 0.94 (note that their study covered new failures); they report a time overhead of 4.6% (considering the coverage computation). For NonDex and iDFlakies the authors conducted extensive studies, but using subjects for which ground truth of flaky tests was not available, so precision cannot be computed. In iDFlakies the time overhead can be fixed *a priori* by the user. The approach leveraging Pattern search makes an evaluation study by randomly sampling tests classified as flaky and ascertaining that they are all correctly classified (which is why we reported precision 1). Finally the Association Rules approach claims a precision of 0.85 and 0.90 over two different projects.

5.6 Threats to validity

Despite our best efforts, our results might still be mined by *threats to validity*. We consider four types of threats [44].

Construct validity. If our empirical experimentation is appropriate to answer the RQs. From different perspectives, RQ1, RQ2 and RQ3 all aim at evaluating FLAST effectiveness in predicting flakiness. In doing this, a potential threat could be choosing wrong metrics that do not properly represent FLAST prediction capability; for example classifiers are typically evaluated by Accuracy, i.e., the ratio between the number of correct predictions and the total number of predictions. In our case though this measure would be misleading, as due to the high proportion of non-flaky tests, it would always provide values close to 1. To prevent this threat, we selected the metrics to use after carefully considering the scope of FLAST, and for the same reason we make available the confusion matrices that provide the full view of prediction results. Another potential threat would be to adopt a misleading validation procedure: to prevent this risk we applied well-known rigorous validations strategies (such as Stratified 10-Fold Cross Validation, and Stratified Shuffle Split with 10 splits). In RQ4 we aim at evaluating FLAST cost: such a study may suffer from many threats, in particular the use of FLAST could be subject to many costs that are hidden or difficult to assess, so that any attempt to evaluate such costs in a laboratory study could be unrealistic. A proper assessment can only be done by putting FLAST in actual production. In this paper we could not deal with this threat, and rather opted to limit the evaluation to directly measurable overheads metrics in terms of execution time and storage requirements. In RQ5, we aim at comparing the performance of FLAST against that of competing approaches. However, the risk of setting an experiment to compare approaches that are actually not comparable against each other is high because, as we explained in Section 4.2, the other existing approaches assume different input information and use different resources. To prevent this threat, we only performed a qualitative comparison over a set of more prominent aspects of the different techniques.

Internal validity. If the observed results are affected by factors different from the treatments. A common internal validity threat lays in the selection of experimental subjects, which we mitigate by adopting data triangulation, executed by gathering data from three distinct datasets available in the literature [4, 22, 34]. Another potential threat descends from trusting such datasets and using them as the ground truth for evaluating FLAST effectiveness. Indeed, if the labeling as flaky or not-flaky were wrong, we might overestimate or under-estimate FLAST effectiveness. If such a threat occurs, we consider that it is most likely that our results might be biased against FLAST, in that as the approaches used in [4, 22, 34] are dynamic, it is more likely that a flaky test is not recognized as such (because by rerunning a test it continues to fail) rather than the vice versa. Other threats may be relative to the parameters set in the application of used algorithms and the accuracy of the measurements themselves: this is mitigated by the application of rigorous *ad-hoc* validation strategies best suited to answer our research questions.

External validity. If, and to what extent, the observed results can be generalized. Our experiments are in line with similar ones present in literature. Additionally, we use for experimentation projects belonging to all three datasets which, to the best of our knowledge, are currently available. As FLAST does not leverage programming language semantics, we do not expect results to drastically vary by considering non-Java subjects. Notwithstanding,

from current observations we cannot draw general conclusions, and more experimentation is needed.

Reliability. If, and to what extent, observations can be reproduced by other researchers. To ensure reproducibility, as said we make available all data and settings related information.

6 USING FLAST IN A CONTINUOUS INTEGRATION ENVIRONMENT

Our results show that FLAST is a simple yet powerful approach for flakiness prediction. Thanks to the simplicity and high-level of abstraction that characterize FLAST functioning, it can be easily and seamlessly adopted in a wide range of industrial and research contexts. Nevertheless, due to its fast and static nature, FLAST results to be exceptionally well suited to be integrated in CI, and in the reminder of this section, we discuss some prominent implications.

Application scenarios. As detailed in Section 3, FLAST can be customized by setting an *ad-hoc* threshold. This threshold embodies the tradeoff between precision and recall of our approach, and can be set to best fit the context in which FLAST is applied. Among others, two main application scenarios can be envisioned. The first application scenario envisages FLAST feedback to be sent back to the test creator, or anyhow to developers, to be directly acted upon via their manual intervention. In this case, it is crucial to ensure the high precision of the approach, as the manual inspection of the output is a costly process and, ultimately, it is also important to ensure the trust of developers in the results of the approach. Under these circumstances, a more conservative threshold (e.g., $\sigma = 0.95$) can be utilized, sacrificing recall for the sake of a higher precision of the test flakiness prediction. In a second application scenario, the output of FLAST is processed automatically, e.g., to refine test case prioritization processes or to determine on an *ad-hoc* basis the number of reruns required to verify if the flaky prediction is true. Given the lower cost of processing false positives under these circumstances, a more encompassing threshold (e.g., $\sigma = 0.5$) can be adopted, so that more flaky tests can be early detected. In addition to these two main application scenarios, FLAST can also be applied in an adaptive way, i.e., with the ability to automatically adjust the threshold based on its past precision. For example, it can start with a more conservative threshold when it is first deployed in the environment, and then, if the precision rates are above some user-defined target, it can relax the threshold a bit in the attempt of reducing the number of false negatives. From time to time FLAST can reevaluate the need to adjust the threshold to maintain its performance within the accepted precision level.

Approaches combination. Even though we showed that its precision is high, FLAST is not intended as an alternative to dynamic approaches (e.g., [4, 22]). FLAST *predicts* if a test is flaky, based on a preexisting ground truth on flaky tests. Dynamic approaches are instead able to *detect* test flakiness by concretely rerunning failing test cases. Our vision is that FLAST provides a remarkably fast, low-cost, and reliable approach to be used *in combination* with dynamic approaches to alleviate the cost of the latter. By predicting with negligible overhead and already at commit time, if a new test is prone to be flaky, FLAST can drastically decrease the percentage of flaky tests that go to the testing stages and hence reduce the many negative impacts of this problem on the development process.

Feedback to developers. The underlying hypothesis on which FLAST is based, i.e. that flaky tests present similar traits, allows the straightforward conversion of data generated by the approach into feedback for developers. In fact, in addition to the precision with which a test is predicted as flaky, it is also possible to provide developers with useful information to support them in fixing flaky tests. From the results of RQ3 we observe that, if information on the nature of flaky tests is available for a project (e.g., by leveraging information stored in issue trackers), FLAST can predict with high precision the flakiness category of a new test case. Additionally, if a test is predicted as flaky, it is possible to instantaneously retrieve examples of similar tests flagged as flaky in the past and, if historical commit data is available, this information can be leveraged to suggest fixes based on how those similar flaky tests were fixed. In the above perspective, we speculate that in the long term using an approach like FLAST can act as a learning-in-the-field tool and will progressively educate developers to recognize typical code patterns and errors that cause flakiness and hence to write more stable tests.

7 CONCLUSIONS

Following the motto *know your neighbor* we proposed a novel approach to predict flaky tests by leveraging test code similarity: test methods whose code is neighbor to that of known flaky tests will also very likely expose flakiness. FLAST has shown to be an effective predictor and to impose very low –actually negligible– time and storage overhead. More importantly, flaky tests can be detected in fully automated way even before they are executed: they can be taken care of before being committed into the test repository, avoiding that testing effort is wasted in rerunning failing tests and code velocity is slowed down waiting for flaky test resolution.

Researchers attention on test flakiness is recent. After a qualitative comparison of existing approaches, we can confidently say that FLAST opens a novel interesting avenue for solving this challenge. Other researchers could propose even better algorithms exploiting test code similarity to prevent a high percentage of flaky tests.

FLAST could be embedded within the adopted IDE or the CI platform, to automatically warn developers against the risk that a new test case or test method might be flaky. While in this paper we have developed and evaluated FLAST, we leave it as a future work direction to develop an integrated environment where it is embedded and evaluated.

Although our study showed that the approach can already be used on small train sizes, another challenge we leave for future work is to devise variants of FLAST acting as more generic predictors that could be used across projects when a train set is not yet available.

As a final remark, FLAST is not to be seen as an alternative to existing dynamic solutions. Rather, we foresee the greatest advantage in using static and dynamic solutions in mutual synergy: FLAST would first prevent many flaky tests by recognizing potentially flaky test code traits. For flaky tests that pass FLAST filtering, these can be detected by dynamic approaches like DeFlaker or even Rerun, but with much less resources. Also this combination of FLAST with dynamic approaches is an important objective for future work.

ACKNOWLEDGMENTS

This research has been motivated and partly supported by a Facebook Research 2019 TAV (Testing and Verification) award.

REFERENCES

[1] Dimitris Achlioptas. 2003. Database-friendly random projections: Johnson-Lindenstrauss with binary coins. *Journal of computer and System Sciences* 66, 4 (2003), 671–687.

[2] Naomi S. Altman. 1992. An Introduction to Kernel and Nearest-Neighbor Nonparametric Regression. *The American Statistician* 46, 3 (1992), 175–185.

[3] Jonathan Bell and Gail Kaiser. 2014. Unit Test Virtualization with VMVM. In *Proceedings of the 36th International Conference on Software Engineering (ICSE 2014)*. ACM, New York, NY, USA, 550–561. https://doi.org/10.1145/2568225.2568248

[4] Jonathan Bell, Owolabi Legunsen, Michael Hilton, Lamyaa Eloussi, Tifany Yung, and Darko Marinov. 2018. DeFlaker: automatically detecting flaky tests. In *Proceedings of the 40th International Conference on Software Engineering*. ACM, 433–444.

[5] Moritz Beller, Georgios Gousios, Annibale Panichella, Sebastian Proksch, Sven Amann, and Andy Zaidman. 2017. Developer testing in the IDE: Patterns, beliefs, and behavior. *IEEE Transactions on Software Engineering* 45, 3 (2017), 261–284.

[6] Jon Louis Bentley. 1975. Multidimensional binary search trees used for associative searching. *Commun. ACM* 18, 9 (1975), 509–517.

[7] Kevin Beyer, Jonathan Goldstein, Raghu Ramakrishnan, and Uri Shaft. 1999. When is "nearest neighbor" meaningful?. In *International conference on database theory*. Springer, 217–235.

[8] Michael Buckland and Fredric Gey. 1994. The relationship between recall and precision. *Journal of the American society for information science* 45, 1 (1994), 12–19.

[9] Emilio Cruciani, Breno Miranda, Roberto Verdecchia, and Antonia Bertolino. 2019. Scalable approaches for test suite reduction. In *41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*. 419–429.

[10] Martin Fowler. 2011. Eradicating non-determinism in tests. https://martinfowler.com/articles/nonDeterminism.html. Accessed: 2019-08-02.

[11] Alessio Gambi, Jonathan Bell, and Andreas Zeller. 2018. Practical test dependency detection. In *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 1–11.

[12] Google Testing Blog. 2008. Testing on the Toilet Avoiding Flaky Tests. https://testing.googleblog.com/2008/04/tott-avoiding-flakey-tests.html. Accessed: 2019-08-06.

[13] Alex Gyori, August Shi, Farah Hariri, and Darko Marinov. 2015. Reliable Testing: Detecting State-polluting Tests to Prevent Test Dependency. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis (ISSTA 2015)*. 223–233.

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

[15] Kim Herzig. 2016. Let's assume we had to pay for testing. Keynote at AST 2016. https://www.kim-herzig.de/2016/06/28/keynote-ast-2016/

[16] Kim Herzig and Nachiappan Nagappan. 2015. Empirically Detecting False Test Alarms Using Association Rules. In *Proceedings of the 37th International Conference on Software Engineering - Volume 2 (ICSE '15)*. 39–48.

[17] William B Johnson and Joram Lindenstrauss. 1984. Extensions of Lipschitz mappings into a Hilbert space. *Contemporary mathematics* 26, 189-206 (1984), 1.

[18] Tariq M King, Dionny Santiago, Justin Phillips, and Peter J Clarke. 2018. Towards a Bayesian Network Model for Predicting Flaky Automated Tests. In *2018 IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C)*. IEEE, 100–107.

[19] Ron Kohavi. 1995. A Study of Cross-validation and Bootstrap for Accuracy Estimation and Model Selection. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence - Volume 2 (IJCAI'95)*. 1137–1143.

[20] Adriaan Labuschagne, Laura Inozemtseva, and Reid Holmes. 2017. Measuring the Cost of Regression Testing in Practice: A Study of Java Projects Using Continuous Integration. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2017)*. 821–830.

[21] Wing Lam, Patrice Godefroid, Suman Nath, Anirudh Santhiar, and Suresh Thummalapenta. 2019. Root Causing Flaky Tests in a Large-scale Industrial Setting. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2019)*. 101–111.

[22] Wing Lam, Reed Oei, August Shi, Darko Marinov, and Tao Xie. 2019. iDFlakies: A Framework for Detecting and Partially Classifying Flaky Tests. In *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*. IEEE, 312–322.

[23] Claire Leong, Abhayendra Singh, Mike Papadakis, Yves Le Traon, and John Micco. 2019. Assessing Transition-based Test Selection Algorithms at Google. In *Proceedings of the 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP '10)*. 101–110.

[24] Jure Leskovec, Anand Rajaraman, and Jeffrey D. Ullman. 2014. *Mining of Massive Datasets*. Cambridge University Press, New York, NY, USA.

[25] Ping Li, Trevor J Hastie, and Kenneth W Church. 2006. Very sparse random projections. In *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 287–296.

[26] Qingzhou Luo, Farah Hariri, Lamyaa Eloussi, and Darko Marinov. 2014. An Empirical Analysis of Flaky Tests. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2014)*. 643–653.

[27] Mateusz Machalica, Alex Samylkin, Meredith Porth, and Satish Chandra. 2019. Predictive Test Selection. In *Proceedings of the 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP '10)*. 91–100.

[28] Atif Memon, Zebao Gao, Bao Nguyen, Sanjeev Dhanda, Eric Nickell, Rob Siemborski, and John Micco. 2017. Taming Google-scale Continuous Testing. In *Proceedings of the 39th International Conference on Software Engineering: Software Engineering in Practice Track (SEIP'17)*. 233–242.

[29] John Micco. 2016. Flaky tests at Google and how we mitigate them. https://testing.googleblog.com/2016/05/flaky-tests-at-google-and-how-we.html. Accessed: 2019-07-22.

[30] John Micco. 2017. The State of Continuous Integration Testing @Google. https://ai.google/research/pubs/pub45880 Accessed: 2019-07-22.

[31] Breno Miranda, Emilio Cruciani, Roberto Verdecchia, and Antonia Bertolino. 2018. FAST approaches to scalable similarity-based test case prioritization. In *40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*. 222–232.

[32] Stephen M Omohundro. 1989. *Five balltree construction algorithms*. International Computer Science Institute Berkeley.

[33] Fabio Palomba and Andy Zaidman. 2017. Does refactoring of test smells induce fixing flaky tests?. In *2017 IEEE international conference on software maintenance and evolution (ICSME)*. IEEE, 1–12.

[34] Fabio Palomba and Andy Zaidman. 2019. The smell of fear: On the relation between test smells and flaky tests. *Empirical Software Engineering* (2019), 1–40.

[35] David Martin Powers. 2011. Evaluation: from precision, recall and F-measure to ROC, informedness, markedness and correlation. (2011).

[36] Kai Presler-Marshall, Eric Horton, Sarah Heckman, and Kathryn T. Stolee. 2019. Wait Wait. No, Tell Me: Analyzing Selenium Configuration Effects on Test Flakiness. In *Proceedings of the 14th International Workshop on Automation of Software Test (AST '19)*. 7–13.

[37] Md Tajmilur Rahman and Peter C. Rigby. 2018. The Impact of Failing, Flaky, and High Failure Tests on the Number of Crash Reports Associated with Firefox Builds. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2018)*. 857–862.

[38] August Shi, Jonathan Bell, and Darko Marinov. 2019. Mitigating the Effects of Flaky Tests on Mutation Testing. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2019)*. 112–122.

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

[40] Pavan Sudarshan. 2012. No more flaky tests on the Go team. https://www.thoughtworks.com/insights/blog/no-more-flaky-tests-go-team. Accessed: 2019-08-06.

[41] Swapna Thorve, Chandani Sreshtha, and Na Meng. 2018. An Empirical Study of Flaky Tests in Android Apps. In *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 534–538.

[42] Arash Vahabzadeh, Amin Milani Fard, and Ali Mesbah. 2015. An Empirical Study of Bugs in Test Code. In *Proceedings of the 2015 IEEE International Conference on Software Maintenance and Evolution (ICSME) (ICSME '15)*. 101–110.

[43] Matias Waterloo, Suzette Person, and Sebastian Elbaum. 2015. Test Analysis: Searching for Faults in Tests. In *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering (ASE '15)*. 149–154.

[44] Claes Wohlin, Per Runeson, Martin Höst, Magnus C. Ohlsson, Bjrn Regnell, and Anders Wessln. 2012. *Experimentation in Software Engineering*. Springer Publishing Company, Incorporated.