IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received April 7, 2021, accepted May 8, 2021, date of publication May 20, 2021, date of current version May 28, 2021.

Digital Object Identifier 10.1109/ACCESS.2021.3082424

Know You Neighbor: Fast Static Prediction of Test Flakiness

ROBERTO VERDECCHIA[1], EMILIO CRUCIANI[2], BRENO MIRANDA[3], AND ANTONIA BERTOLINO[4]

[1]Department of Computer Science, Vrije Universiteit Amsterdam, 1081 HV Amsterdam, The Netherlands
[2]Computerwissenschaften, Universität Salzburg, 5020 Salzburg, Austria
[3]Center of Informatics, Federal University of Pernambuco, Recife 50670-901, Brazil
[4]Istituto di Scienza e Tecnologie dell'Informazione "A. Faedo", Consiglio Nazionale delle Ricerche, 56124 Pisa, Italy

Corresponding author: Antonia Bertolino (antonia.bertolino@isti.cnr.it)

This work was supported in part by the Facebook Research 2019 Testing and Verification Award.

ABSTRACT *Context:* Flaky tests plague regression testing in Continuous Integration environments by slowing down change releases and wasting testing time and effort. Despite the growing interest in mitigating the burden of test flakiness, how to efficiently and effectively detect flaky tests is still an open problem. *Objective:* In this study, we present and evaluate FLAST, an approach designed to statically predict test flakiness. FLAST leverages vector-space modeling, similarity search, dimensionality reduction, and k-Nearest Neighbor classification in order to timely and efficiently detect test flakiness. *Method:* In order to gain insights into the efficiency and effectiveness of FLAST, we conduct an empirical evaluation of the approach by considering 13 real-world projects, for a total of 1,383 flaky and 26,702 non-flaky tests. We carry out a quantitative comparison of FLAST with the state-of-the-art methods to detect test flakiness, by considering a balanced dataset comprising 1,402 real-world flaky and as many non-flaky tests. *Results:* From the results we observe that the effectiveness of FLAST is comparable with the state-of-the-art, while providing considerable gains in terms of efficiency. In addition, the results demonstrate how by tuning the threshold of the approach FLAST can be made more conservative, so to reduce false positives, at the cost of missing more potentially flaky tests. *Conclusion:* The collected results demonstrate that FLAST provides a fast, low-cost and reliable approach that can be used to guide test rerunning, or to gate the inclusion of new potentially flaky tests.

INDEX TERMS Software testing, flaky tests, prediction, static analysis, similarity.

I. INTRODUCTION

Flaky tests *can intermittently pass or fail even for the same code version* [1]. Flakiness hinders regression testing in many ways [2]–[5], 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. Indeed, in Google, almost 16% of individual tests contain some form of flakiness [6], 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) [7]. 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 [8]. 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 [9]. 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 [10].

Test flakiness wastes 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, e.g., even up to 10 times [5], [11], to ascertain that failures are not intermittent.

Approaches have been proposed to reduce the rerunning overhead, e.g., by using code analysis [12], [13], so that even one only execution might be sufficient. In the following

The associate editor coordinating the review of this manuscript and approving it for publication was Antonio Piccinno.

we denote flakiness detection techniques that rely on test execution (one or more times) as *dynamic* techniques. In contrast, other researchers have recently proposed to recognize flakiness based on characteristics of known flaky tests, e.g., [14]–[16]. These *static* approaches either rely on manual identification of such characteristics by experts [14], [16], or learn them from a vast historical dataset [15]. In either case they would require large effort to be generalized.

As wished by Harman and O'Hearn from Facebook [17], research should find a quick yet effective method for test flakiness assessment. Indeed, Ziftci and Cavalcanti from Google [18] state that "*it is important to fix flaky tests as quickly as possible to keep development velocity high*". Accordingly, our aim is a method that can timely detect a flaky test even before it is executed.

In previous studies [19], [20] we have shown that test code similarity can provide an effective instrument for test suite prioritization and reduction. Inspired by such studies, in this work we *leverage test code similarity for identifying flaky tests*. This is also in line with a recent empirical study [21] in which standard Machine Learning classifiers have been used to identify a "vocabulary" of flaky tests. We propose the FLAST approach that uses a k-Nearest Neighbor classifier [22] to predict the flaky/non-flaky nature of any new test.

We report the results of evaluating the effectiveness and efficiency of FLAST over 13 projects utilized in a previous work on test flakiness [21]. Notably, FLAST could detect flaky tests with a mean value for precision varying from 0.69 (with 0.48 recall) to 0.83 (with 0.22 recall). Precision values depend on a threshold σ we can tune to make the approach more or less conservative: by increasing the threshold value the percentage of false positives can be decreased (i.e., higher precision), but at the cost of missing more potentially flaky tests (i.e., lower recall).

FLAST inherently relies on other detectors for obtaining an initial set of flaky tests on which it can be trained. However our evaluation shows that it can reach a good precision already in projects having quite few flaky tests. In fact, FLAST is not proposed as an alternative to existing dynamic techniques, on the contrary static and dynamic analysis could be used in synergy. A straightforward scenario would be that FLAST guides the usage of dynamic tools, e.g., the effort spent with the latter could be prioritized toward those tests that FLAST predicts as flaky. Another deployment route for FLAST could be to proactively gate flaky tests from being included in the test repository. FLAST could be embedded in the used CI framework, e.g., Travis CI,[1] and as new or modified test cases are committed, it could automatically (and at negligible cost) identify potentially flaky tests and send them back to developers.

Summarizing, we provide the following contributions:

- *Idea:* a novel tunable method for similarity-based flakiness detection based on static analysis of test code.
- *Evaluation:* a study over 13 projects including in total more than 28K test methods, ∼1,400 of which flaky.
- *Replication package:* a replication package[2] including a proof-of-concept implementation of FLAST, along with the entirety of the data used for the empirical evaluation, the scripts to replicate the evaluation, and all intermediate and final results obtained. Additionally, in the replication package, we provide a documentation of the preliminary experimentation used to tune the approach, supported by the utilized scripts, data, and gathered results.

In the next section we overview related work and in Section III we describe the FLAST approach. The planning of the empirical evaluation, including goal, experiment material, design and procedure, is presented in Section IV. Results are reported in Section V and thoroughly discussed in Section VI. Finally, we outline potential applications of FLAST in CI practice in Section VII and draw conclusions in Section VIII.

II. RELATED WORK

In recent years flaky tests are drawing increasing researchers' attention, also triggered by practitioners' alerts about the relevance and spread of the problem [3], [6], [8], [17], [23]. This overview of related work is based on a thorough search of literature.[3]

Many empirical studies have been conducted aiming at better understanding the nature and extent of test flakiness (e.g., [1], [9], [23]–[31]). Both studies by Luo *et al.* [1] and Vahabzadeh *et al.* [24] examined the causes of flaky tests over the central repository of the Apache Software Foundation, whereas Thorve *et al.* [26] conducted a similar study over Android apps. Other studies aim at assessing the relevance and cost of the problem, including the work by Labuschagne *et al.* [9] over 61 projects from GitHub Archive, and the one by Rahman and Rigby [25] who studied how many of the crash reports submitted by Firefox users were associated with flaky tests, while Eck *et al.* [28] asked developers from Mozilla to classify previously fixed flaky tests. The study by Kowalczyk *et al.* [23] at Apple introduces and evaluates a quantitative notion of flakiness. Lam *et al.* [30] focus on the under-explored class of non-order-dependent flaky tests, showing that actually they may still depend on test execution ordering. Another study by Lam *et al.* [31] investigates at which commit flakiness is introduced, and finds that 85% of flaky tests are already so when added or modified, but the remaining 15% ones become flaky later at unrelated commits.

Few work aim at preventing flaky tests by early detecting potential causes of flakiness. Gyori *et al.* [32] propose the POLDET technique that timely notices if a new added test case "pollutes" the state of the shared heap or the file system, so to possibly cause other tests to intermittently fail.

[1] https://travis-ci.org

[2] https://github.com/FlakinessStaticDetection/FLAST

[3] More precisely, we launched a generic query searching for string {*"flaky test" OR "flaky tests" OR flakiness*} over ACM DL, IEEExplore and Scopus on 26 February 2021.

TABLE 1. Qualitative comparison among existing flakiness detection techniques.

Approach	Analysis type	SUT coverage	Flakiness type	Scope	Action type	Expert knowledge	Training
Rerun [6]	Dynamic	No	Generic	Subset	Reactive	No	No
DeFlaker [12]	Dynamic	Yes	Generic	Subset	Proactive	No	No
iDFlakies [13]	Dynamic	No	Generic	Subset	Proactive	No	No
NonDex [38]	Dynamic	No	Specific	All	Proactive	Yes	No
SHAKER [39]	Dynamic	No	Generic	All	Proactive	No	Yes
FLASH [40]	Dynamic	Yes	Specific	All	Proactive	Yes	Yes
Association Rules [15]	Static	No	Generic	Subset	Reactive	No	Yes
Pattern Search [14]	Static	No	Specific	All	Proactive	Yes	Yes
Bayesian Network [16]	Static	No	Generic	All	Proactive	Yes	Yes
ML Classifiers [21]	Static	No	Generic	All	Proactive	No	Yes
FLAST	Static	No	Generic	All	Proactive	No	Yes

Gambi *et al.* [33] develop the PRADET approach, which discovers test dependencies by flow analysis and iterative testing. Finally, Parry et al. [34] present the FITTER technique that leverages automated program repair to expose and help fixing latent flakiness that could be induced by order dependency or resource leaks.

Other authors aim at identifying flakiness root causes. Both the RootFinder tool by Lam *et al.* [8] and the DIVERGENCE technique by Ziftci *et al.* [18] execute the flaky tests after instrumentation and then compare the logs of passing and failing runs. Terragni *et al.* [35] propose instead an approach that reruns the flaky tests within different execution environments, or ''clusters'', without any need to use instrumentation.

Different directions are pursued to support debugging and repairing of different types of flaky tests. Shi *et al.* [36] present the iFixFlakies framework that supports the automated fixing of flakiness caused by test order-dependency. Malm *et al.* [37] instead develop an approach that automatically identifies the delays in test code and helps determine the best wait time to avoid timing-related flakiness.

The work that is more closely related with ours is those relative to flaky test detection. The state-of-practice approach, referred to as Rerun [12], consists of 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 [6], [11]. Researchers aim at approaches that can improve on Rerun, which is costly and not very precise.[4]

In Table 1 we summarize schematically the comparison of FLAST against Rerun and the nine competitor approaches (listed in the first column) we found from our search of literature. The comparison is conducted along the following dimensions (reported from the second to the eighth column of Table 1):

- *Analysis type*: Static (no test execution needed) or Dynamic (test must be executed at least once)
- *SUT coverage*: Yes (approach uses coverage reports of the system under test) or No otherwise
- *Flakiness type*: Generic (approach targets any type of flaky test) or Specific (only some specific types of flakiness can be detected)
- *Scope*: All (approach is applied to all tests) or Subset (only a part of tests is analyzed)
- *Action type*: Proactive (approach actively searches for flaky tests) or Reactive (approach is invoked only in reaction to transitions)
- *Expert knowledge*: Yes (approach needs expert consultancy) or No otherwise
- *Training set*: Yes (approach needs to be trained on a set of known flaky tests) or No otherwise

Rerun [6] and five more approaches [12], [13], [38]–[40] are *dynamic*, while four approaches [14]–[16], [21] are static, as is ours (*2nd column*).

Only one approach, viz. DeFlaker, relies on code coverage reports (*3rd column*): indeed, collecting coverage may be quite costly in CI practice [41] and because of this the authors of DeFlaker propose a lightweight technique leveraging differential coverage. We also marked Yes in this column for FLASH that is a domain-specific technique addressing probabilistic and Machine Learning (ML) applications. It aims at detecting intermittent failure of test assertions due to improper assignment of thresholds, and hence it needs to instrument the code for monitoring the distribution of assertion values.

With few exceptions, most approaches can detect generic type of flaky tests (*4th column*). NonDex focuses on flakiness due to ADINS (Assumes a Deterministic Implementation of a Non-deterministic Specification) code. Pattern Search is proposed as an approach to detect pre-determined types of test code faults, among which timing dependency. Finally, as above explained, FLASH addresses test assertion flakiness in probabilistic and ML applications.

Not all approaches are applied to every test case, as we do (*5th column*). Rerun,[5] 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.

[4]For example in the study by Pinto *et al.* [21] a good percentage of tests labeled as flaky passed 99 times (out of 100 reruns) and failed in only one case, whereas in practice only a few reruns, e.g., up to 10, are routinely done.

[5]Although in theory all tests could be repeated a number of times regardless of their outcome, the high cost of doing so prevents this in practice, and Rerun is commonly applied only to those tests that produce a transition from the previous testing cycle.

Almost all approaches, but Rerun and Association Rules, take action in proactive way for detecting flaky tests (*6th column*).

A critical feature is whether an approach is fully automated, or otherwise it requires manual effort and expert knowledge for customization/preparation (*7th column*), which may clearly hinder their practical adoption. The latter is the case for NonDex, FLASH, Pattern Search, and Bayesian Network. In contrast FLAST, as well as Rerun, DeFlaker, iDFlakies, SHAKER, Association Rules, and ML Classifiers do not require any human consultancy.

Finally more than half of the approaches requires a training phase, as does FLAST (*8th column*).

Based on the above analysis, the work most similar to ours is the one using ML classifiers: indeed both propose to statically detect flakiness by supervised learning from a sample of test cases. However, we can identify notable differences both in the objectives and in the approach taken. Concerning the respective objectives, in the work of Pinto *et al.* [21] the authors conduct an empirical study to identify a "vocabulary" of flaky tests, which could be leveraged to predict flakiness. Our objective, instead, is to provide an approach for predicting flakiness within a Continuous Integration environment (as we discuss in Section VII).

Due to the difference in objectives, the two approaches are quite different: Pinto *et al.* [21] apply five state-of-the-art ML classifiers, using standard available implementations. In contrast, FLAST develops a novel enhanced implementation of a Nearest Neighbors classifier, introducing the concept of a *threshold*, which permits to tune precision vs. recall, and a *dimensionality reduction* technique, which allows for drastically improving efficiency. We explain such enhancements in the next section.

III. 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 $\ell : T \rightarrow \{0, 1\}$ 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: we model the tests in T as points in some vector space, where we fix a notion of source-code similarity and dissimilarity among test cases, and then train on T a variation of a k-Nearest Neighbor classifier [22].

Since FLAST is designed to be used in a Continuous Integration (CI) environment, where the test suite rapidly evolves over time, the choice of the classification technique is crucial. At each new addition or deletion of tests, classifiers belonging to the "eager learners" category would need to rebuild their models in order to use the new knowledge, and such an operation could be costly. Instead, "lazy learners" as k-Nearest Neighbor are much more suitable for this scenario since they do not need to build a model, but can directly learn from data, thus having the capacity of immediately exploiting new knowledge.

In the rest of the section we explain in detail the kind of representation we choose for the tests and the algorithm that we use to predict tests as flaky.

A. VECTOR SPACE MODELING

Similarly to what we did in a previous work [20], we model the tests in T as points in an n-dimensional vector space using the *bag-of-words model* [42]: each test case t is represented as the multiset (i.e., a set that allows multiple instances of its elements) of the lowercase tokens composing its source code, split by whitespace characters and punctuation. We purposely decided not to manipulate the input data, e.g., we did not exclude comments, as they are exclusively the original ones written by developers and not added by researchers a posteriori, reflecting the real-world nature of our experimental subjects. As shown by Pinto *et al.* [21], further manipulating the input data (e.g., stopwords removal, stemming, include/exclude Java identifier, and others) does not provide any concrete improvement in the effectiveness of the classification. A concrete example of the tokenization process is reported in Listing 1-3, where we consider the source code of two flaky test methods (Listing 1-2) and a non-flaky one (Listing 3). The tokens shared among all three of the test methods are reported in bold (brown), while the tokens shared exclusively between the two flaky test methods in italic (violet).

According to the model created via the tokenization process, 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 using the Term-Frequency (TF) scheme, i.e., according to the multiplicity of i among the tokens of t. By considering the three test methods presented in Listings 1, 2, and 3, we can observe in Table 2, Column "ID", how the tokenization process leads to the identification of 21 distinct tokens (i.e., the dimension of the vector space), whose occurrences vary among the three test cases (as reported in the Column "Token Occurrences"). The choice of TF instead of other more common and complex weighting schemes such as Term-Frequency Inverse-Document-Frequency (TF-IDF), where frequent terms such as "assert" that appear in most tests would be penalized, is crucial for the use of our approach in CI. By using TF-IDF, the weights of a single test would depend on the entire test suite (in order to count the global

```
public void singleVerifier()
  HttpRequest request1 = get("localhost").trustAllHosts();
  HttpRequest request2 = get("localhost").trustAllHosts();
  assertNotNull(
    ((HttpsURLConnection) request1.getConnection()).getHostnameVerifier());
  assertNotNull(
    ((HttpsURLConnection) request2.getConnection()).getHostnameVerifier());
  assertEquals(
    ((HttpsURLConnection) request1.getConnection()).getHostnameVerifier(),
    ((HttpsURLConnection) request2.getConnection()).getHostnameVerifier());
}
```

Listing 1. Example test method 1 (flaky).

```
public void singleSslSocketFactory() {
  HttpRequest request1 = get("localhost").trustAllCerts();
  HttpRequest request2 = get("localhost").trustAllCerts();
  assertNotNull(
    ((HttpsURLConnection) request1.getConnection()).getSSLSocketFactory());
  assertNotNull(
    ((HttpsURLConnection) request2.getConnection()).getSSLSocketFactory());
  assertEquals(
    ((HttpsURLConnection) request1.getConnection()).getSSLSocketFactory(),
    ((HttpsURLConnection) request2.getConnection()).getSSLSocketFactory());
}
```

Listing 2. Example test method 2 (flaky).

```
public void dotLookup() {
  Context context = Context.newContext("String");
  assertNotNull(context);
  assertEquals("String", context.get("."));
}
```

Listing 3. Example test method 3 (non-flaky).

TABLE 2. Bag-of-words representations of the example test methods (Listings 1, 2, and 3).

ID	Token	Token Occurrences		
		TM 1	TM 2	TM 3
t01	assertEquals	1	1	1
t02	assertNotNull	2	2	1
t03	context	0	0	5
t04	dotLookup	0	0	1
t05	get	2	2	1
t06	getConnection	4	4	0
t07	getHostnameVerifier	4	0	0
t08	getSSLSocketFactory	0	4	0
t09	localhost	2	2	0
t10	HttpRequest	2	2	0
t11	HttpsURLConnection	4	4	0
t12	newContext	0	0	1
t13	public	1	1	1
t14	request1	3	3	0
t15	request2	3	3	0
t16	singleSslSocketFactory	0	1	0
t17	singleVerifier	1	0	0
t18	String	0	0	1
t19	trustAllCerts	0	2	0
t20	trustAllHosts	2	0	0
t21	void	1	1	1

frequency of a term): hence, whenever the test suite evolves and new tests are added the vector representation of the entire test suite would change accordingly. Instead, when using TF as we do, a new test does not have any impact on the vector representation of old ones.

B. 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* $S_c(\boldsymbol{s}, \boldsymbol{t}) = \cos\theta = \frac{\langle \boldsymbol{s}, \boldsymbol{t} \rangle}{\|\boldsymbol{s}\| \cdot \|\boldsymbol{t}\|}$, whereby $\langle \boldsymbol{s}, \boldsymbol{t} \rangle = \sum_{i=1}^{n} \boldsymbol{s}_i \boldsymbol{t}_i$ 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 dissimilarity* $D_c(\boldsymbol{s}, \boldsymbol{t}) = 1 - S_c(\boldsymbol{s}, \boldsymbol{t})$.

C. DIMENSIONALITY REDUCTION

When working in high dimensional spaces, distances do not behave as we expect due to the "curse of dimensionality": the volume of the space increases much faster than the density of points as the dimensionality increases, making the data look very sparse; as a consequence distances tend to become uniform, i.e., both pairs of similar and dissimilar data points are far away [43]. As commonly done for the kind of classification task we are facing, in order to mitigate such an unwanted effect we apply a dimensionality reduction technique called *sparse random projection* [44], [45]. Roughly speaking, points are projected onto a random d-dimensional subspace of $\mathbb{R}^n$, with $d = \Omega\left(\frac{\log |T|}{\varepsilon^2}\right)$, such that the pairwise distance of the projected points is preserved up to a multiplicative factor $(1 \pm \varepsilon)$; this fact is known as the Johnson-Lindenstrauss Lemma [46]. Note that the same random projection also approximately preserves the pairwise angles between points up to the same multiplicative factor [47], hence their pairwise cosine similarity/dissimilarity. Such a reduction in the dimensionality, other than mitigating the curse of dimensionality with a positive impact on the effectiveness of our approach, also allows to obtain immediate gains in terms of efficiency while performing the neighbors search. Efficiency can be crucial in extremely-large-scale scenarios where test suites are comprised of millions of tests [41], [48].

The dimensionality d of the random subspace onto which points are projected is independent of the initial dimensionality n, i.e, from the content of the tests, and much smaller than n, since the number of tests $|T|$ is typically smaller than their dimensionality n and the dimensionality d after the projection is much smaller than $|T|$. The value of ε, i.e., the distortion of the distances after the projection, can be customized to yield a different effectiveness/efficiency tradeoff in the distance measurement.

A visual representation of dimensionality reduction via random projection is depicted in Figure 1, considering the three test methods reported in Listings 1, 2, and 3.

Once again, we underline that our approach is designed to work in CI environments, where the test suite changes over time. In fact, whenever tests are added to the test suite, the vector representation of the old tests is not affected as well as their projection. This is essentially due to the TF weighting scheme we adopted, that sets to 0 the components relative to the new tokens introduced by the additions. Such a property

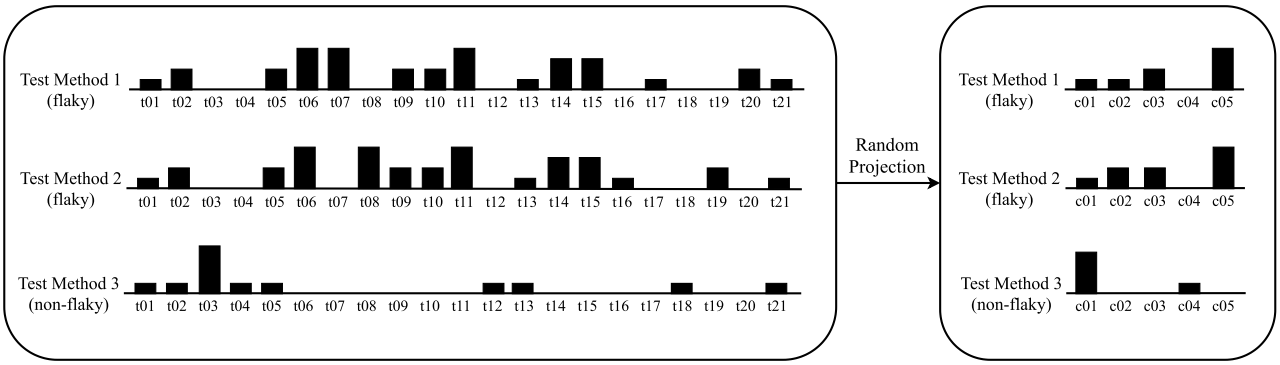


FIGURE 1. Visual representation of random projections of the example test methods (Listings 1, 2, and 3).

Algorithm 1 FLAST Prediction

Input: test suite T; flakiness indicator $\ell : T \rightarrow \{0, 1\}$; test to predict 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** $\frac{\phi_s}{\phi_s + \psi_s} \geqslant \sigma$: **return** True ▷ predict the test as *flaky*
5: **else**: **return** False ▷ predict the test as *non-flaky*

allows us to only project the newly added tests, without altering the projection of the previous ones. The pairwise distance of the points could be affected, but in negligible way for any practical scenario. A more detailed discussion is deferred to the Appendix.

D. 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, 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).

The flakiness prediction performed by FLAST is sketched in Algorithm 1. First, the unknown test case s is mapped to a vector $\boldsymbol{s}$ and projected onto the same vector space used for the tests in T, as discussed in the previous subsection on dimensionality reduction. Then, FLAST searches for the set N_s of k neighbor tests, that are closest to $\boldsymbol{s}$ according to the cosine dissimilarity of their vector representations (Line 1); in our implementation we look for the neighbors via a naive linear search, but the description in Line 1 is mathematical

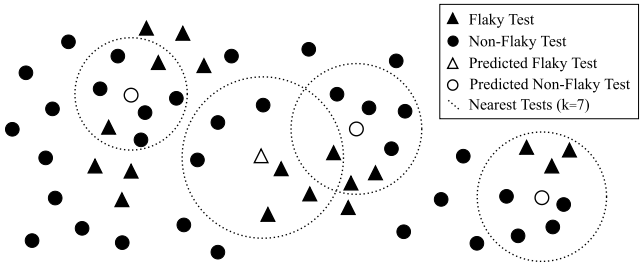


FIGURE 2. Visual representation of FLAST prediction.

rather than procedural since the same operation could be done using other data structures and algorithms (e.g., using space partitioning data structures). 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 dissimilarity to $\boldsymbol{s}$. Test s is predicted to be *flaky* if $\frac{\phi_s}{\phi_s + \psi_s} \geq \sigma$, 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.

We provide a visual representation of the intuition behind FLAST prediction in Figure 2. 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.

E. PRECISION TUNING

The threshold parameter $\sigma \in [0, 1]$ in Algorithm 1 has been introduced to make the approach more flexible, and in particular to be able to trade-off the precision of FLAST against its recall. Note that in order for the threshold to have an effect on the prediction, we need to have $k > 1$. Precisely, with reference to Algorithm 1 (Line 4), $\sigma = 0.5$ corresponds to using a default threshold, i.e., we predict a test s as flaky if $\phi_s \geqslant \psi_s$, which is the standard for a k-Nearest Neighbor classifier. Using instead values of σ higher than 0.5 corresponds

to requiring higher confidence in predicting a test as flaky, i.e., to higher precision values. This increase in precision is obtained though at the cost of a lower recall. At the extreme value of $\sigma = 1$ FLAST would predict a test as flaky only if all its k nearest neighbors are flaky (independently of their distance). Values smaller than 0.5 would instead intuitively increase the recall at the cost of a lower precision, with the extreme value of $\sigma = 0$ that would result in a dummy classifier that predicts everything as flaky (with recall=1, but low precision); we do not cover these values.

IV. EXPERIMENTAL PLANNING

A. GOALS

The research objective of this experiment is to: *Analyze* the performance of FLAST *for the purpose of* evaluation *with respect to* its ability of predicting test flakiness *from the point of view of* potential users *in the context of* continuous integration environments. In order to achieve this objective, we defined the following goals for this study (following the Goal/Question/Metric method by Basili *et al.* [49]):

- $\mathbf{G}_1$ Evaluate the *effectiveness* and *efficiency* of FLAST.
- $\mathbf{G}_2$ Analyze FLAST for the purpose of comparison with a state-of-the-art technique with respect to effectiveness and efficiency.

The assessment of the goal $\mathbf{G}_1$ is performed by the following research questions ($\mathbf{Q}_1$ and $\mathbf{Q}_2$) and metrics ($\mathbf{M}_1$ to $\mathbf{M}_5$).

- $\mathbf{Q}_1$ How *effective* is FLAST in predicting test flakiness?

$$P = \frac{TP}{TP + FP} \quad (\boldsymbol{M}_1)$$
$$R = \frac{TP}{TP + FN} \quad (\boldsymbol{M}_2)$$

where TP, FP, and FN respectively denote *true positive*, *false positive*, and *false negative* predictions, i.e., the correct flaky predictions, the incorrect flaky predictions, and the incorrect non-flaky predictions respectively. Both P and R values are in the range [0, 1].

$\mathbf{M}_1$ measures the fraction of correct flaky predictions over the total number of flaky predictions, i.e., it could be considered as a measure of FLAST's *correctness*. $\mathbf{M}_2$ measures the fraction of correct flaky predictions over the total number of flaky tests, so it could instead be thought of as a measure of FLAST's *completeness*.

- $\mathbf{Q}_2$ How *efficient* is FLAST in predicting test flakiness?

$$TT = \textit{training time} \quad (\boldsymbol{M}_3)$$
$$PT = \textit{prediction time} \quad (\boldsymbol{M}_4)$$
$$SO = \textit{storage overhead} \quad (\boldsymbol{M}_5)$$

$\mathbf{M}_3$ measures the time, in seconds, needed by FLAST to vectorize the test suite, apply dimensionality reduction, and store the vectors in the data structure used to search for the nearest neighbors. $\mathbf{M}_4$ measures the average time, in milliseconds, needed by FLAST to vectorize a new test, apply dimensionality reduction, and perform the nearest neighbors search in order to predict the nature of a new test. $\mathbf{M}_5$ measures the size in memory and on disk, in megabytes, of the vector representation of the tests of the entire test suite.

The assessment of the goal $\mathbf{G}_2$ is performed by the research question ($\mathbf{Q}_3$) and metrics $\mathbf{M}_1$ to $\mathbf{M}_5$—introduced before for $\mathbf{G}_1$.

- $\mathbf{Q}_3$ What is the performance of FLAST when compared to a state-of-the-art approach with respect to *effectiveness* and *efficiency* in predicting test flakiness?

For selecting a state-of-the-art competitor to FLAST we refer to the approaches examined in Table 1. As hinted in the introduction and later discussed in Section VII, dynamic detectors are seen as complementary rather than as alternative to FLAST. Among the static approaches, we cannot use the Association Rules [15] as they have been mined from an ultra-large set of test alarms available in Microsoft and are hardly generalizable outside their original context. We also exclude Pattern Search [14] and Bayesian Network [16] as both require expert knowledge in contrast with FLAST that is fully automated. Hence the ML classifiers investigated by Pinto *et al.* [21] appear to provide the fairest choice for an empirical comparison of effectiveness and efficiency.

B. EXPERIMENTAL MATERIAL

1) EVALUATION DATASET

a: DATASET USED TO ANSWER Q_1 AND Q_2

In order to answer Q_1 and Q_2, we leverage the test suites of the software projects gathered by Pinto *et al.* [21]. Their dataset was constructed based on the DeFlaker[6] benchmark. The authors complemented the DeFlaker dataset, maintaining the information on flaky tests, by rerunning 100 times the test suites of each project in the most recent version present in GitHub at the time of the study; test cases that had a consistent outcome across all executions (e.g., the test passes 100 times) were flagged as non-flaky. The dataset is accessible in a replication package available online.[7] Note that the tests labeled as flaky could come from different versions of each software project [12], while the tests labeled as non-flaky all come from the same version (the last at the time of rerun) of each software project [21].

It is important to note that the identification of a test as non-flaky *via* reruns has to be considered only an estimate as, while observing an intermittent behavior is sufficient to label a test as flaky, rerunning can never guarantee non-flakiness, regardless of a high number of reruns. However, we can consider a non-flaky estimate based on 100 reruns as a practical trade-off.

In order to answer Q_1 and Q_2, we are interested in a scenario which resembles the closest an *in vivo* one. We consider the software projects of the dataset by Pinto *et al.* [21] separately and in their entirety. In particular, differently from Pinto *et al.*, we do not merge test suites of different projects and we do not sample tests to balance the number of flaky

[6] www.deflaker.org/icsecomp/

[7] https://github.com/damorimRG/msr4flakiness/

and non-flaky ones. In order to have the bare minimum information for our experimental evaluation, i.e., enough data for training and testing the approach, we included in the extended dataset all the projects from the work of Pinto *et al.* [21] that contain at least 4 flaky tests. Due to the approach followed in the work of Pinto *et al.* [21] to build the dataset, it could happen that a same test method is included twice, both labeled as flaky (because it was originally labeled so in the DeFlaker dataset) and as non-flaky. This duplication could be either due to the fact that the 100 reruns could not reproduce the intermittent behavior observed in DeFlaker, or because the test method has been modified and Pinto *et al.* actually rerun a different version. We decided to remove from the non-flaky set the 26 test methods that were fully identical to a flaky one (in package, name, and content), but left the modified ones in the dataset. Overall, we take into account the test suites of 13 different software projects for a total of 28,085 tests, out of which 1,383 flaky and 26,702 non-flaky. More detailed information on which projects have been selected and on their number of flaky and non-flaky tests are provided in Tables 3 and 4.

b: DATASET USED TO ANSWER Q_3

To evaluate the performance of FLAST with respect to the ML classifiers [21], we use the same data configuration employed by Pinto *et al.* to assess their effectiveness. In actual projects, the number of non-flaky tests is normally much higher than the number of flaky tests. In [21], to mitigate issues arising when learning from imbalanced data, the authors opted to train their models by considering the whole set of flaky tests (1,402[8]) and a sample of an equal number of non-flaky tests: the resulting balanced dataset is henceforth referred to as `Pinto-DS`. Therefore, concerning Q_3, we decided to adopt this `Pinto-DS` dataset in order to perform a fair comparison and to be able to directly compare our results with those reported in [21].

c: REMARK REGARDING THE DATASETS

Given that all included projects are Java-based, in our evaluation we consider a ''test'' to be a ''Java test method''. Nevertheless, we remark that FLAST only leverages the syntactical structure of the test and would work out of the box for any programming language and any definition of test.

2) REPLICATION PACKAGE

With the aim of supporting independent verification and replication of the experiments, we make available a replication package hosted on GitHub.[9] It includes a proof-of-concept implementation of FLAST, along with the entirety of the data used for the empirical evaluation, the scripts to replicate the evaluation, and all intermediate and final results obtained. Additionally, in the replication package, we provide a documentation of the preliminary experimentation used to tune the approach, supported by the utilized scripts, data, and gathered results.

3) HARDWARE

All experiments were run on a 2015 MacBook Pro with a 2.7 GHz Intel Core i5 processor, 8 GB 1867 Mhz DDR3 memory, running macOS Catalina 10.15.6.

C. EXPERIMENT DESIGN

1) TO ANSWER Q_1

From a preliminary experimentation carried out for this study, reported in the online replication package, we observed how FLAST behaves by changing the setting of its parameters and the size of the training set. This led us to identify four different configurations of parameters of FLAST in our empirical evaluation. Specifically we consider two different values of σ, namely a default threshold $\sigma = 0.5$ and a more conservative threshold $\sigma = 0.95$, and two different values for the number k of nearest neighbors equal to 7 and 3. The use of a higher threshold allows to obtain higher precision values at the cost of lower recall; the use of a low value of k (i.e., $k = 3$, namely the smallest odd value of k allowing the use of a threshold) provides low bias and high variance in the classification, while increasing it decreases the variance and increases the bias making the approach more robust to noise. Note also that the higher value of k (i.e., $k = 7$) could not be increased much due to the extremely low number of flaky tests in some of the projects (sometimes even less than 7 in the training set). We set the dimensionality d of the tests after the random projection such that the distortion $\varepsilon = 0.3$.

We infer $\boldsymbol{M}_1$ and $\boldsymbol{M}_2$ through a Stratified Shuffle Split Cross Validation with 30 splits, using a random 20% subset as test set and the remaining as training set, separately for each considered combination of parameters. Stratification 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 [50]. Moreover, we guarantee that each fold contains at least one flaky test both in the training and in the testing sets.

2) TO ANSWER Q_2

We measure $\boldsymbol{M}_3$, $\boldsymbol{M}_4$, and $\boldsymbol{M}_5$ as average times and storage size of FLAST among the runs of a Stratified Shuffle Split Cross Validation, with 30 splits and using 20% of the dataset as test set and the remaining as training set, separately for each considered combination of parameters k and σ (same as for Q_1).

3) TO ANSWER Q_3

Our goal is to compare the effectiveness and efficiency of FLAST against the MS classifiers used by Pinto *et al.* [21].

As for effectiveness, we apply our approach FLAST over the `Pinto-DS` dataset, and compare our results in terms of precision and recall with those achieved on this same dataset

[8] An attentive reader may notice that this number is higher than the sum of flaky tests exposed by the 13 projects we consider in Table 3. This is because as said we did not consider projects having less than 4 flaky tests.

[9] https://github.com/FlakinessStaticDetection/FLAST

TABLE 3. FLAST's effectiveness.

Project	#Flaky	#Non-Flaky	Precision ($\sigma = 0.5$, $k = 7$)	Recall ($\sigma = 0.5$, $k = 7$)	Precision ($\sigma = 0.95$, $k = 7$)	Recall ($\sigma = 0.95$, $k = 7$)	Precision ($\sigma = 0.5$, $k = 3$)	Recall ($\sigma = 0.5$, $k = 3$)	Precision ($\sigma = 0.95$, $k = 3$)	Recall ($\sigma = 0.95$, $k = 3$)
achilles	67	8	0.98	1.00	1.00	1.00	1.00	1.00	1.00	1.00
alluxio	4	2,737	1.00	0.43	-	0.00	0.95	0.63	-	0.00
ambari	4	8	1.00	0.43	-	0.00	1.00	0.60	1.00	0.20
hadoop	305	2,984	0.88	0.70	0.97	0.34	0.87	0.73	0.96	0.57
jackrabbit	8	9,215	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
jimfs	7	448	0.75	0.93	1.00	0.37	0.70	0.77	1.00	0.83
ninja	18	937	0.88	0.60	-	0.00	0.74	0.64	0.84	0.37
okhttp	66	1,482	0.93	0.48	1.00	0.19	0.77	0.53	1.00	0.34
oozie	856	314	1.00	1.00	1.00	0.98	1.00	1.00	1.00	0.99
oryx	13	372	0.60	0.26	-	0.00	0.63	0.51	1.00	0.10
spring-boot	14	6,807	1.00	0.44	-	0.00	1.00	0.74	1.00	0.09
togglz	11	387	0.00	0.00	-	0.00	0.00	0.00	0.00	0.00
wro4j	10	1,003	0.00	0.00	-	0.00	0.00	0.00	0.00	0.00
Mean	106	2,054	0.69	0.48	0.83	0.22	0.67	0.55	0.73	0.35
Median	13	937	0.88	0.44	1.00	0.00	0.77	0.63	1.00	0.20

The values k and σ are parameters of FLAST (see Algorithm 1). The values reported in the table are mean values (among defined values) of a Stratified Shuffle Split Cross Validation. Undefined values are reported as "-" and are obtained whenever FLAST did not predict any test as flaky, also implying 0.00 recall values. Mean and Median values are only computed among the defined values and, for #Flaky and #Non-Flaky, are rounded to the closest integer.

by the competitor ML classifiers as reported in [21, Table 4]. The effectiveness results for FLAST were obtained by using the same combinations of k and σ used for Q_1 and Q_2.

As for efficiency, the study in [21] does not report any data. We note that four of the five classifiers studied in the work of Pinto *et al.* [21] (namely Random Forest, Decision Tree, Naive Bayes and Support Vector) are "eager learners" and hence, as we explained in Section III, they are not apt for application in a CI scenario. The fifth classifier considered by Pinto *et al.* [21] is a standard implementation of Nearest Neighbors, belonging to the category of "lazy learners" as ours. For fairness, we hence compare the results of FLAST against those obtained by applying a Nearest Neighbor classifier, because it is the only approach investigated by Pinto *et al.* [21] that is a suitable competitor for a CI environment.

Precision and recall results of FLAST, as well as training time, prediction time, and storage overhead, are computed as average values among the runs of a Stratified Shuffle Split Cross Validation, with 30 splits and using 20% of the dataset as test set and the remaining as training set, separately for each considered combination of parameters k and σ.

D. EXPERIMENTAL PROCEDURE

The k-Nearest Neighbors search used by FLAST is implemented through the `python` library `sickie-learn`.[10] The implementation of the competitor Nearest Neighbor approach [21], used to compare its efficiency with that of FLAST, utilizes the same library and the original parameters.

Training time and prediction time are measured via the `python` function `time.perf_counter()`. Storage overhead is measured by serializing the data structure used to store in memory the vector representation of the test suite (`numpy.ndarray`) via the `python` function `pickle.dump()` and then measuring its size on disk via the `python` function `os.path.getsize()`.

[10]https://scikit-learn.org

V. ANALYSIS

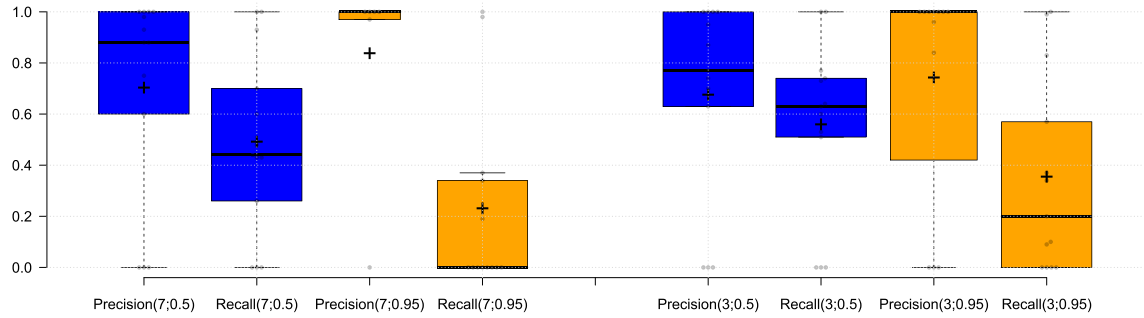
A. EFFECTIVENESS (Q_1)

The box plots of Figure 3 show the distribution of precision and recall values obtained by FLAST when applied over the evaluation dataset. For these metrics, the higher the result (reported in the vertical axis), the better. The left plot depicts the precision and recall achieved by FLAST when considering $k = 7$, whereas the right one depicts the results for $k = 3$. For each metric displayed (precision or recall), the left (blue) boxes refer to the results for the scenario with threshold $\sigma = 0.5$, while the right (orange) boxes refer to the scenario

TABLE 4. FLAST's efficiency.

Project	#Tests	Storage (MB)	Training (s)	Prediction (ms)
achilles	75	0.29	0.05	1.32
alluxio	2,741	19.30	0.82	0.79
ambari	12	0.03	0.01	1.85
hadoop	3,289	23.68	1.16	0.86
jackrabbit	9,223	74.89	2.57	0.94
jimfs	455	2.48	0.15	0.79
ninja	955	5.83	0.26	0.71
okhttp	1,548	10.12	0.59	0.84
oozie	1,170	7.35	0.41	0.88
oryx	385	2.04	0.39	1.79
spring-boot	6,821	53.53	2.42	0.99
togglz	398	2.12	0.14	0.87
wro4j	1,013	6.23	0.28	0.73
Mean	2,160	15.99	0.71	1.03
Median	950	6.23	0.39	0.87

The efficiency has been measured as average values of a Stratified Shuffle Split Cross Validation. The results are then reported as average among the combinations of k and σ. Mean and Median values for #Tests are rounded to the closest integer.



Precision and recall of FLAST with $\sigma = 0.5$ (blue) and with $\sigma = 0.95$ (orange). The four boxplots on the left depict the results when considering $k = 7$, whereas the four on the right consider $k = 3$.

FIGURE 3. Boxplots representing FLAST's effectiveness (Table 3).

TABLE 5. Comparison with state of the art (effectiveness).

Approach	Precision	Recall
Random Forest	**0.99**	0.91
Decision Tree	0.89	0.88
Naive Bayes	0.93	0.80
Support Vector	0.93	0.92
Nearest Neighbor	0.97	0.88
FLAST ($k = 7, \sigma = 0.5$)	0.92	0.91
FLAST ($k = 7, \sigma = 0.95$)	**0.99**	0.72
FLAST ($k = 3, \sigma = 0.5$)	0.93	**0.93**
FLAST ($k = 3, \sigma = 0.95$)	0.98	0.84

Values for ML classifiers are reported from [21, Table 4]. Values for FLAST are computed on `Pinto-DS` dataset, the same dataset used in [21], as a result of a Shuffle Split Cross Validation with a test set size of 80%, as in [21]. `Pinto-DS` is comprised of 1,402 flaky and 1,402 non-flaky tests.

TABLE 6. Comparison with state of the art (efficiency).

Approach	Storage (MB)	Training (s)	Prediction (ms)
Nearest Neighbor	474.80	1.16	2.67
FLAST	**19.81**	**0.95**	**0.58**

Efficiency results are computed as a result of a Shuffle Split Cross Validation on `Pinto-DS`, dataset comprised of 1,402 flaky and 1,402 non-flaky tests. Since efficiency results are not reported in [21], Nearest Neighbor's efficiency has been measured by implementing it using the same library of FLAST, for a fair comparison. Reported values for FLAST are computed as average among the combinations of k and σ.

with threshold $\sigma = 0.95$. A detailed breakdown of precision and recall values per threshold and per project is available in Table 3. The number of flaky and non-flaky tests per project is also available from the same table, Columns 2 and 3, respectively.

B. EFFICIENCY (Q_2)

Table 4 depicts the performance of FLAST in terms of training time (M_3), prediction time (M_4), and storage overhead (M_5). A detailed breakdown per project, including the total number of test cases used in the evaluation (second column), is available from the same table.

C. COMPARISON WITH STATE-OF-THE-ART (Q_3)

The results of the comparison in effectiveness and efficiency between FLAST and Pinto *et al.*'s ML classifiers are reported in Tables 5 and 6. Table 5 summarizes the effectiveness of the approaches with respect to precision (M_1) and recall(M_2). Table 6 reports the efficiency results of the approaches with respect to training time (M_3), prediction time (M_4), and storage overhead (M_5).

VI. DISCUSSION

A. EVALUATION OF RESULTS AND IMPLICATIONS

1) EFFECTIVENESS (Q_1)

When considering the consolidated results for $k = 7$ and the less conservative scenario ($\sigma = 0.5$), an overall average precision of 0.69 was obtained, i.e., when FLAST classified a test as flaky it got it right 69% of the times. Median precision results are higher (0.88) since FLAST does not have good precision only on three projects, namely `jackrabbit`, `wro4j`, and `togglz`; such results are analyzed and discussed in detail later in this section, in a dedicated paragraph, through a manual inspection of false positive predictions. When we increase the threshold to $\sigma = 0.95$, thus making FLAST behave in a more conservative way, the precision on every project increases resulting in an overall average precision of 0.83.

Regarding recall, instead, FLAST achieved an overall average of 0.48, i.e., FLAST correctly identified 48% of the flaky tests, when using a threshold $\sigma = 0.5$. When increasing the threshold to $\sigma = 0.95$, we observe a tradeoff between precision and recall: as expected, the former increases while the latter decreases. Indeed, the overall average recall decreases to 0.22. Note that these low mean (and, specially, median) recall values, besides to the more conservative behavior of FLAST using threshold $\sigma = 0.95$, are due also to the fact that many projects contain only few flaky tests. Indeed, for 7 projects for which FLAST got 0.00 recall, FLAST did not predict any test as flaky, i.e., obtained undefined precision (equal to "-" in Table 3) and, thus, 0.00 recall (since the number of true positive is 0); we also note that the same 7 projects are among those with the lowest number of flaky tests in the dataset (all with less than 20 flaky tests). It is unsurprising that the high level of conservativeness due to the threshold $\sigma = 0.95$ together with the low number of flaky tests on which FLAST is trained make predicting a test

```
public void shouldNotWatchForChangeUnlessCheckCompleted() {
  Context.get().getConfig()
    .setResourceWatcherUpdatePeriod( 1 );
  final CacheKey key = new CacheKey(GROUP_NAME,
    ResourceType.JS, true);
  when(mockResourceWatcher.tryAsyncCheck(Mockito.eq(key)))
    .thenReturn(false);
  victim.get(key);
  assertFalse(victim.wasCheckedForChange(key));
  when(mockResourceWatcher.tryAsyncCheck(Mockito.eq(key)))
    .thenReturn(true);
  victim.get(key);
  assertTrue(victim.wasCheckedForChange(key));
}
```

Listing 4. Example of flaky test (parameter change).

```
public void shouldNotWatchForChangeUnlessCheckCompleted() {
  Context.get().getConfig()
    .setResourceWatcherUpdatePeriod( 100 );
  final CacheKey key = new CacheKey(GROUP_NAME,
    ResourceType.JS, true);
  when(mockResourceWatcher.tryAsyncCheck(Mockito.eq(key)))
    .thenReturn(false);
  victim.get(key);
  assertFalse(victim.wasCheckedForChange(key));
  when(mockResourceWatcher.tryAsyncCheck(Mockito.eq(key)))
    .thenReturn(true);
  victim.get(key);
  assertTrue(victim.wasCheckedForChange(key));
}
```

Listing 5. Example of non-flaky test (parameter change).

as flaky very unlikely. On the other hand, for the 3 projects having the lowest number of flaky tests, namely `alluxio`, `ambari`, and `jimfs` (respectively 4, 4 and 7 flaky tests), with $\sigma = 0.5$ FLAST could achieve precision values of 1, 1 and 0.75, respectively.

Thus a related important question would be to know how much data FLAST would need to start making good predictions. We explore this question in the online replication package.

When using $k = 3$, the number of projects for which it makes predictions in the more conservative scenario increases from 6 to 9. When using the threshold $\sigma = 0.5$, the overall average precision and recall are 0.67 and 0.55, respectively. For the threshold $\sigma = 0.95$ the average precision is 0.73, while the recall achieved is 0.35.

Manual Per-Project Inspection of False Positives:

In order to gain further insights into the results of our empirical evaluation, we conducted a manual inspection on the false positive predictions of FLAST. We considered the three projects for which FLAST had lowest precision (considering $k = 7$ and $\sigma = 0.5$), namely `jackrabbit`, `wro4j`, and `togglz`, where FLAST wrongly predicted as flaky two, three, and seven distinct tests respectively.

We noted that most of the mispredictions have been influenced by the existence of extremely similar or even identical tests (due to duplicate code and/or small changes across different versions as explained in Section IV-B) labeled as flaky in the dataset. In particular, we found three distinct scenarios in which FLAST did not predict flakiness correctly because the syntax of the test alone was not sufficient to identify its flaky nature.

In the first scenario, we observe how the modification of even a single parameter of a test could change the nature of the test from being flaky to non-flaky. An example coming from `wro4j` is that reported in Listings 4 and 5, where the flakiness of the test in Listing 4 seems to be due to the update period of 1 second, probably too frequent given that it has been updated to 100 seconds in a later version of the test, reported in Listing 5, that has not been recognized as flaky anymore.

In the second scenario, we observe how bad developer habits could increase the similarity between flaky and non-flaky tests. As an example we consider two tests coming from `togglz` project and reported in Listings 6 and 7, where the developer has probably produced the later test by copying and pasting the other. Indeed, by observing the listings, the content of the non-flaky test is essentially a subset of the content of the flaky one, except for the first setup instruction.

In the third scenario, we observe how tests could be identical in syntax even if different in behavior, given that tests preconditions are not taken into account by FLAST being syntactically located in a different area of the source code. An example of this setting can be found in two tests coming from `jackrabbit`: the test methods are fully identical, but they actually belong to different Java classes and, thus, have different preconditions, i.e., one has global restrictions while performing read operations while the other does not. The tests can be found, as well as the other previously discussed tests, in our replication package linked in Section IV-B.

> **Main findings Q_1 (Effectiveness):** Median precision of FLAST ranged from 0.77 to 1.00 and median recall ranged from 0.20 to 0.63. By tuning the threshold of the approach, FLAST can be made more conservative, so to reduce false positives, at the cost of missing more potentially flaky tests.

2) EFFICIENCY (Q_2)

As we can see in Table 4, overall, the average training time observed was less than one second (∼0.71 s), and the average prediction time roughly one millisecond (∼1.03 ms). The average storage overhead accounts for roughly 16 MB with a median of ∼6 MB (with an average project size of ∼2,000 test methods). When considering the project with the highest number of test cases, `jackrabbit` with 9,223 tests, FLAST takes 2.57 s for training. For the sake of fairness it should be noted that such costs do not consider the needed effort of identifying the original set of flaky tests on which FLAST is trained.

In the experiments, FLAST used 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 $\mathcal{O}(|T|)$ (considering k constant, as in our experiments) to predict the nature of a test $s \notin T$. 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

```
public void testRemovingOfActivationStrategy()
    throws ConnectionException {

  FeatureState savedFeatureState =
    new FeatureState(TestFeature.FEATURE);
  savedFeatureState.setStrategyId
    (UsernameActivationStrategy.ID);
  savedFeatureState.setParameter(
    UsernameActivationStrategy.PARAM_USERS,
    "user1, user2, user3");
  stateRepository.setFeatureState(savedFeatureState);
  FeatureState loadedFeatureState =
    stateRepository.getFeatureState(TestFeature.FEATURE);
  assertThat(reflectionEquals(
    savedFeatureState, loadedFeatureState),
    is(true));

  // save same feature, but without activation strategy. should remove an existing one
  FeatureState featureStateWithoutStrategy = new FeatureState(TestFeature.FEATURE);
  stateRepository.setFeatureState(featureStateWithoutStrategy);
  loadedFeatureState = stateRepository.getFeatureState(TestFeature.FEATURE);
  assertThat(reflectionEquals(featureStateWithoutStrategy, loadedFeatureState), is(true));
}
```

Listing 6. Example of flaky test (duplicate code).

```
public void testActivationStrategySavingAndLoading()
    throws Exception {
  setupTestWithEmptyDatastore();
  FeatureState savedFeatureState =
    new FeatureState(TestFeature.FEATURE);
  savedFeatureState.setStrategyId
    (UsernameActivationStrategy.ID);
  savedFeatureState.setParameter(
    UsernameActivationStrategy.PARAM_USERS,
    "user1, user2, user3");
  stateRepository.setFeatureState(savedFeatureState);
  FeatureState loadedFeatureState =
    stateRepository.getFeatureState(TestFeature.FEATURE);
  assertThat(reflectionEquals(
    savedFeatureState, loadedFeatureState),
    is(true));
}
```

Listing 7. Example of non-flaky test (duplicate code).

`kd-tree` [51] and `balltree` [52] or in approximate ways using techniques such as Locality Sensitive Hashing [42], that could be of interest to further speed up the computation of the nearest neighbor search. As observed in Table 4, the current way used by FLAST to look for nearest neighbors can be considered efficient, specially w.r.t. the scale of the software projects considered in the experiments. However, at larger scales where test suites are comprised of millions of tests [41], [48], the use of more sophisticated approaches should be considered to keep the approach efficient.

Main findings Q_2 (Efficiency): FLAST is a lightweight approach with very low operational cost. It is efficient in terms of both time and storage. For the projects considered in our evaluation (∼2,000 test methods) the median training and prediction time were 0.39 seconds and 0.87 milliseconds, respectively, with a storage overhead of 6.23 MB.

3) COMPARISON WITH STATE-OF-THE-ART (Q_3)

Analyzing the effectiveness results reported in Table 5 we can see that, among the ML classifiers, the highest precision is obtained by Random Forest (0.99) while the highest recall is obtained by Support Vector (0.92). With respect to the combinations of k and σ of FLAST, the highest precision (0.99) is obtained with $k = 7$ and $\sigma = 0.95$, while the highest recall (0.93) is obtained with $k = 3$ and $\sigma = 0.5$. Overall, FLAST ($k = 7$ and $\sigma = 0.95$) and Random Forest achieve the highest precision, whereas FLAST ($k = 3$ and $\sigma = 0.5$) produces the highest recall.

With respect to efficiency, FLAST outperforms the Nearest Neighbor competitor in all of the considered metrics, with a comparable training time (about 1 s on a dataset with 2,804 tests for both), an average prediction time 4.5 times smaller (about 0.6 ms for FLAST vs more than 2.5 ms for Nearest Neighbor), and a storage overhead almost 24 times smaller (less than 20 MB for FLAST vs almost 475 MB for Nearest Neighbor).

Regarding the efficiency of FLAST, the advantage in prediction time could be seen as negligible at the scale of small/medium software projects as those considered in our comparison, since it is in the order of 1 ms. However, at large scale, where such predictions could be repeated millions of times, having a faster predictor could play an important role.

Regarding storage, instead, the improvement of FLAST due to the use of sparse random projections [44], [45] is immediately noticeable, allowing the use of roughly 24 times less storage on disk and in memory when predicting new tests.

Main findings Q_3 (Comparison with SOTA): The effectiveness of FLAST is comparable with the state-of-the-art, while providing considerable gains in terms of efficiency: in our evaluation, FLAST used less than 5% the storage overhead that would be required by competitor approach while making predictions 4.6 times faster.

B. THREATS TO VALIDITY

Despite our best efforts, our results might still be undermined by *threats to validity*. We consider four types of threats [53].

1) CONSTRUCT VALIDITY

If our empirical experimentation is appropriate to answer the research questions. Concerning Q_1 a potential threat could be choosing a wrong metric that does not properly represent FLAST's 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, for effectiveness we selected precision (M_1) after carefully considering the scope of FLAST. Another potential threat would be to adopt a misleading validation procedure: to prevent this risk we applied a rigorous validations strategy, namely Stratified Shuffle Split Cross Validation. With Q_2 we aim at evaluating FLAST's efficiency: 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. Finally, with Q_3 we aim at comparing the effectiveness and efficiency of FLAST with 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 show in Section II, most other existing approaches assume different input information and use different resources. To prevent this threat, we preceded the experiment with a qualitative comparison (in Section II) over a set of more prominent dimension and then choose the most similar work.

2) INTERNAL VALIDITY

If the observed results are affected by factors different from the treatments. A common internal validity threat lays in the selection of the experimental subjects, which we mitigate by gathering data from one robust dataset available in the literature [21]. Nevertheless, due to the process followed to establish it, for some tests two versions could have been included, one of which flaky and the other one non-flaky (cfr. Section IV-B). While this could have influenced the observed results, we do not deem this as a major threat to validity, as we verified that this duplication affects only a minute portion of the dataset. In addition, to mitigate this threat, the identified tests that resulted to be identical and marked as flaky and non-flaky in the two versions were removed from the dataset. More in general, a potential threat descends from trusting such a dataset and using it as ground truth for evaluating FLAST's effectiveness. Indeed, if the labeling of test cases as flaky or not-flaky were wrong, we might over-estimate or under-estimate FLAST's effectiveness. If such a threat occurred, we consider that it is most likely that our results might be biased against FLAST, in that, as the classification is done dynamically, 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.

With reference to the study conducted to answer (Q_3), the sampling of non-flaky tests to build the `Pinto-DS` dataset might have introduced a bias. However as we aimed at a fair comparison against the ML classifiers in [21], that was an obligatory choice, and this did not affect in any way our results relative to (Q_1 and Q_2).

Other internal validity 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 a study of parameters variations we performed (reported in the replication package) and by the application of rigorous *ad-hoc* validation strategies best suited to answer our research questions.

3) EXTERNAL VALIDITY

If, and to what extent, the observed results can be generalized. FLAST is evaluated over a set of 13 projects (Q_1 and Q_2) and also over the `Pinto-DS` dataset (Q_3). Although our study is in line with similar studies in literature, and uses a very accurate dataset in terms of number of reruns used to identify flaky/non-flaky tests, we are aware that our experimentation may not be sufficient for drawing generally valid conclusions beyond the examined subjects. To address this threat, FLAST should be studied on more projects with flaky tests detected by reruns, which requires an extensive effort for establishing the ground truth for the experiments. Further studies should also consider different non-Java subjects, even though FLAST does not leverage programming language semantics, and so we do not expect results to drastically vary.

4) RELIABILITY

If, and to what extent, observations can be reproduced by other researchers. To ensure reproducibility, we make available all data and settings related information in our replication package.

VII. USING FLAST IN A CONTINUOUS INTEGRATION ENVIRONMENT

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

A. COMBINING STATIC AND DYNAMIC APPROACHES

FLAST is not intended as an alternative to dynamic approaches (e.g., [12], [13]). 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, and improve their efficiency. Indeed, FLAST can *predict* if a test is flaky, based on a preexisting ground truth on flaky tests. However, even though we showed that its precision is high, it could still provide false positives. Dynamic approaches instead can *detect* test flakiness by concretely rerunning failing test cases. By predicting with negligible overhead, and already at commit time, which tests are prone to be flaky, FLAST can guide dynamic tools, e.g., to refine test case prioritization processes or to indicate which tests should be rerun a higher number of times. We expect that the usage of FLAST in combination with dynamic approaches would help to more efficiently spend the testing budget.

B. GATING FLAKY TESTS

FLAST could be also used to impede that flaky tests are added to the repository. In fact, it could be integrated within the CI platform in use and be run before new tests are committed: if any among the new tests look "suspiciously" similar to previously detected flaky tests, FLAST feedback is sent back

to the test creator to be directly acted upon for possible fixing of flakiness before the tests enter the continuous testing cycles. However, for such a deployment scenario become practical, it would be important that FLAST is also able to explain why a test was predicted as flaky. This is in fact a direction of our future work, which we expand below.

C. FEEDBACK TO TESTERS

The underlying hypothesis on which FLAST is based, i.e., that flaky tests present similar traits, allows the conversion of data generated by the approach into feedback for testers. In fact, in addition to predicting a test as flaky, it is also possible to provide testers with specific information to support them in fixing flakiness. For example, if flaky tests are classified into categories according to their causes (as in [1], [28]), FLAST could also predict the flakiness category of each suspicious test. Also, when a test is predicted as flaky and returned to its creator, FLAST could retrieve examples of similar tests flagged as flaky in the past and, if historical commit data are 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.

D. CUSTOMIZATION

As discussed in Section III, by tuning the threshold σ we can vary the trade-off between precision and recall of FLAST's results. This flexibility can be used to best fit the context in which FLAST is applied, depending on the importance of false positives *vs.* false negatives.

In the scenario of combining static and dynamic flakiness detectors, test cases predicted as flaky are then automatically processed by the test execution engine, with the goal of confirming FLAST's prediction. Under such circumstances, a more encompassing threshold (e.g., $\sigma = 0.5$) should be adopted, which would give a higher recall and hence allow testers to early identify more potentially flaky tests.

The gating scenario instead would envisage that FLAST's feedback would be acted upon by testers via manual intervention. In this case, it is crucial to ensure the highest precision of the approach, as the manual inspection of the output is a costly process and, ultimately, it is important to gain the trust of testers in the results of the approach by preventing as much as possible false positives. In this scenario, a more conservative threshold (e.g., $\sigma = 0.95$) should be utilized, sacrificing recall for the sake of more precise predictions.

In addition, we speculate that FLAST could also be applied in an adaptive way, i.e., with the ability to automatically adjust the threshold σ. For example, the tool could be set to a more conservative threshold when it is first deployed in the environment, and then, as the precision rates grow above some user-defined target, it could adaptively relax the threshold in the aim of reducing the number of false negatives. By monitoring its own performance, FLAST can be empowered to reevaluate the need to adjust the threshold to maintain its precision within an acceptable bound.

VIII. CONCLUSION

Following the motto *know you 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 guide dynamic tools or 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 size training sets, 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 training set is not yet available. An interesting extension of FLAST could be also to explore the idea of enhancing our notion of similarity, currently based on cosine similarity, to consider also broader information, such as, e.g., potential test smells associated with flakiness, as well as other hints we got from our inspection of false positives.

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 detect many flaky tests by recognizing potentially flaky test code traits. For flaky tests that pass FLAST's filtering, these can still 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.

APPENDIX. ON THE RANDOM PROJECTION IN CI

As briefly discussed in Section III, whenever new tests are added to the test suite, as in CI environments, our approach does not require to re-project the entire test suite since the old tests remain unaltered in the random subspace.

Formally, let $X \in \mathbb{R}^{n \times |T|}$ be the matrix of the test suite T, with the columns of X being the vectors representing the

tests, and let $R \in \mathbb{R}^{d \times n}$ be the random projection matrix. It follows that $Y = RX \in \mathbb{R}^{d \times |T|}$ is the matrix of the projected tests, namely the columns of Y are the d-dimensional vectors representing the tests after the random projection.

Consider the addition of a set S of new tests containing a total of m new tokens. At this point, we need to "augment" the old vectors w.r.t. the m new tokens by adding to each of their previous vector representations m new rows containing 0, since the m tokens are new w.r.t. the previous ones, while the previous remain the same thanks to the TF weighting scheme. Moreover, we need to consider the vector representation of the new tests in S. Formally, let $\bar{X} \in \mathbb{R}^{(n+m)\times(|T|+|S|)}$ be the matrix of the new test suite $T \cup S$ in the new $(n+m)$-dimensional space. The columns of $\bar{X}$ are the augmented vectors, i.e., for all $j \in \{1, \ldots, |T|\}$ (the columns relative to old tests) we have that $\bar{X}_{ij} = X_{ij}$ for $i \in \{1, \ldots, n\}$ (the components relative to the old tokens remain the same) and $\bar{X}_{ij} = 0$ for $i \in \{n+1, \ldots, n+m\}$ (the components relative to the new tokens are set to 0); for all $j \in \{|T|+1, \ldots, |T|+|S|\}$ we have the columns relative to the new tests in S. Let $\bar{R} \in \mathbb{R}^{d \times (n+m)}$ be the augmented projection matrix, i.e., for all $j \in \{1, \ldots, n\}$ (the columns relative to old tokens) we have $\bar{R}_{ij} = R_{ij}$ for all i; for all $j \in \{n+1, \ldots, n+m\}$ we have the columns relative to the m new tokens in S, which are completely new. It follows that $\bar{Y} = \bar{R}\bar{X} \in \mathbb{R}^{d \times (|T|+|S|)}$ is the matrix of the tests that have been projected in the same d-dimensional random subspace.

Note that, for all i and for all $j \in \{1, \ldots, n\}$, it holds that

$$\bar{Y}_{ij} = \sum_{l=1}^{n+m} \bar{R}_{il}\bar{X}_{lj} \overset{(a)}{=} \sum_{l=1}^{n} \bar{R}_{il}\bar{X}_{lj} \overset{(b)}{=} \sum_{l=1}^{n} R_{il}X_{lj} = Y_{ij},$$

where, by definition of $\bar{X}$ and $\bar{R}$: in (a) we use that $\bar{X}_{lj} = 0$ for $l > n$, and in (b) we use that $\bar{R}_{il} = R_{il}$ and $\bar{X}_{lj} = X_{lj}$ for $l \leq n$. In other words, by calling y a column of Y and $\bar{y}$ the corresponding column of $\bar{Y}$, it follows that $\bar{y} = y$, i.e., applying the old projection matrix to the old vectors is equivalent to applying the new projection matrix to their augmented version. Therefore, whenever a set S of new tests is added to T the only operation needed is to augment the projection matrix R and only project S in the random subspace, since the new projection matrix does not alter the projection of T.

REFERENCES

[1] Q. Luo, F. Hariri, L. Eloussi, and D. Marinov, "An empirical analysis of flaky tests," in *Proc. 22nd ACM SIGSOFT Int. Symp. Found. Softw. Eng.*, New York, NY, USA, 2014, pp. 643–653.

[2] Google Testing Blog. (2008). *Testing on the Toilet Avoiding Flaky Tests*. Accessed: Aug. 6, 2019. [Online]. Available: https://testing.googleblog.com/2008/04/tott-avoiding-flakey-tests.html

[3] M. Fowler. (2011). *Eradicating Non-Determinism in Tests*. Accessed: Aug. 2, 2019. [Online]. Available: https://martinfowler.com/articles/nonDeterminism.html

[4] P. Sudarshan. (2012). *No More Flaky Tests on the Go Team*. Accessed: Aug. 6, 2019. [Online]. Available: https://www.thoughtworks.com/insights/blog/no-more-flaky-tests-go-team,

[5] MDN Web Docs Mozilla. (2019). *Test Verification*. Accessed: Jul. 13, 2020. [Online]. Available: https://developer.mozilla.org/en-US/docs/Mozilla/QA/Test_Verification

[6] J. Micco. (2016). *Flaky Tests at Google and How Mitigate Them*. Accessed: Jul. 22, 2019. [Online]. Available: https://testing.googleblog.com/2016/05/flaky-tests-at-google-and-how-we%.html

[7] C. Leong, A. Singh, M. Papadakis, Y. L. Traon, and J. Micco, "Assessing transition-based test selection algorithms at Google," in *Proc. 41st Int. Conf. Softw. Eng., Softw. Eng. Pract.*, Piscataway, NJ, USA, Mar. 2019, pp. 101–110.

[8] W. Lam, P. Godefroid, S. Nath, A. Santhiar, and S. Thummalapenta, "Root causing flaky tests in a large-scale industrial setting," in *Proc. 28th ACM SIGSOFT Int. Symp. Softw. Test. Anal.*, New York, NY, USA, Jul. 2019, pp. 101–111.

[9] A. Labuschagne, L. Inozemtseva, and R. Holmes, "Measuring the cost of regression testing in practice: A study of java projects using continuous integration," in *Proc. 11th Joint Meeting Found. Softw. Eng.*, New York, NY, USA, Aug. 2017, pp. 821–830.

[10] M. Beller, G. Gousios, A. Panichella, S. Proksch, S. Amann, and A. Zaidman, "Developer testing in the IDE: Patterns, beliefs, and behavior," *IEEE Trans. Softw. Eng.*, vol. 45, no. 3, pp. 261–284, Mar. 2017.

[11] J. Micco. (2017). *The State of Continuous Integration Testing Google*. Accessed: May 20, 2021. [Online]. Available: https://ai.google/research/pubs/pub45880

[12] J. Bell, O. Legunsen, M. Hilton, L. Eloussi, T. Yung, and D. Marinov, "DeFlaker: Automatically detecting flaky tests," in *Proc. 40th Int. Conf. Softw. Eng.*, New York, NY, USA, May 2018, pp. 433–444.

[13] W. Lam, R. Oei, A. Shi, D. Marinov, and T. Xie, "IDFlakies: A framework for detecting and partially classifying flaky tests," in *Proc. 12th IEEE Conf. Softw. Test., Validation Verification (ICST)*, Apr. 2019, pp. 312–322.

[14] M. Waterloo, S. Person, and S. Elbaum, "Test analysis: Searching for faults in tests (N)," in *Proc. 30th IEEE/ACM Int. Conf. Automated Softw. Eng. (ASE)*, Piscataway, NJ, USA, Nov. 2015, pp. 149–154.

[15] K. Herzig and N. Nagappan, "Empirically detecting false test alarms using association rules," in *Proc. IEEE/ACM 37th IEEE Int. Conf. Softw. Eng.*, May 2015, pp. 39–48.

[16] T. M. King, D. Santiago, J. Phillips, and P. J. Clarke, "Towards a Bayesian network model for predicting flaky automated tests," in *Proc. IEEE Int. Conf. Softw. Qual., Rel. Secur. Companion (QRS-C)*, Jul. 2018, pp. 100–107.

[17] M. Harman and P. O'Hearn, "From start-ups to scale-ups: Opportunities and open problems for static and dynamic program analysis," in *Proc. IEEE 18th Int. Work. Conf. Source Code Anal. Manipulation (SCAM)*, Sep. 2018, pp. 1–23.

[18] C. Ziftci and D. Cavalcanti, "De-flake your tests: Automatically locating root causes of flaky tests in code at Google," in *Proc. IEEE Int. Conf. Softw. Maintenance Evol. (ICSME)*, Sep. 2020, pp. 736–745.

[19] B. Miranda, E. Cruciani, R. Verdecchia, and A. Bertolino, "FAST approaches to scalable similarity-based test case prioritization," in *Proc. 40th Int. Conf. Softw. Eng.*, New York, NY, USA, May 2018, pp. 222–232.

[20] E. Cruciani, B. Miranda, R. Verdecchia, and A. Bertolino, "Scalable approaches for test suite reduction," in *Proc. IEEE/ACM 41st Int. Conf. Softw. Eng. (ICSE)*, May 2019, pp. 419–429.

[21] G. Pinto, B. Miranda, S. Dissanayake, M. d'Amorim, C. Treude, and A. Bertolino, "What is the vocabulary of flaky tests?" in *Proc. 17th Int. Conf. Mining Softw. Repositories*, Jun. 2020, pp. 492–502.

[22] N. S. Altman, "An introduction to kernel and nearest-neighbor nonparametric regression," *Amer. Statist.*, vol. 46, no. 3, pp. 175–185, Aug. 1992.

[23] E. Kowalczyk, K. Nair, Z. Gao, L. Silberstein, T. Long, and A. Memon, "Modeling and ranking flaky tests at Apple," in *Proc. ACM/IEEE 42nd Int. Conf. Softw. Eng., Softw. Eng. Pract.*, New York, NY, USA, Jun. 2020, pp. 110–119.

[24] A. Vahabzadeh, A. M. Fard, and A. Mesbah, "An empirical study of bugs in test code," in *Proc. IEEE Int. Conf. Softw. Maintenance Evol. (ICSME)*, Washington, DC, USA, Sep. 2015, pp. 101–110.

[25] M. T. Rahman and P. C. Rigby, "The impact of failing, flaky, and high failure tests on the number of crash reports associated with firefox builds," in *Proc. 26th ACM Joint Meeting Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng.*, New York, NY, USA, Oct. 2018, pp. 857–862.

[26] S. Thorve, C. Sreshtha, and N. Meng, "An empirical study of flaky tests in Android apps," in *Proc. IEEE Int. Conf. Softw. Maintenance Evol. (ICSME)*, Washington, DC, USA, Sep. 2018, pp. 534–538.

[27] K. Presler-Marshall, E. Horton, S. Heckman, and K. Stolee, "Wait, wait. No, tell Me. Analyzing selenium configuration effects on test flakiness," in *Proc. IEEE/ACM 14th Int. Workshop Autom. Softw. Test (AST)*, Piscataway, NJ, USA, May 2019, pp. 7–13.

[28] M. Eck, F. Palomba, M. Castelluccio, and A. Bacchelli, "Understanding flaky tests: The developer's perspective," in *Proc. 27th ACM Joint Meeting Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng.*, Aug. 2019, pp. 830–840.

[29] W. Lam, K. Muälu, H. Sajnani, and S. Thummalapenta, "A study on the lifecycle of flaky tests," in *Proc. ACM/IEEE 42nd Int. Conf. Softw. Eng.*, New York, NY, USA, Jun. 2020, pp. 1471–1482.

[30] W. Lam, S. Winter, A. Astorga, V. Stodden, and D. Marinov, "Understanding reproducibility and characteristics of flaky tests through test reruns in java projects," in *Proc. IEEE 31st Int. Symp. Softw. Rel. Eng. (ISSRE)*, Oct. 2020, pp. 403–413.

[31] W. Lam, S. Winter, A. Wei, T. Xie, D. Marinov, and J. Bell, "A large-scale longitudinal study of flaky tests," *Proc. ACM Program. Lang.*, vol. 4, no. 5, pp. 1–29, Nov. 2020.

[32] A. Gyori, A. Shi, F. Hariri, and D. Marinov, "Reliable testing: Detecting state-polluting tests to prevent test dependency," in *Proc. Int. Symp. Softw. Test. Anal.*, New York, NY, USA, Jul. 2015, pp. 223–233.

[33] A. Gambi, J. Bell, and A. Zeller, "Practical test dependency detection," in *Proc. IEEE 11th Int. Conf. Softw. Test., Verification Validation (ICST)*, Apr. 2018, pp. 1–11.

[34] O. Parry, G. M. Kapfhammer, M. Hilton, and P. Mcminn, "Flake it 'Till you make it: Using automated repair to induce and fix latent test flakiness," in *Proc. IEEE/ACM 42nd Int. Conf. Softw. Eng. Workshops*, New York, NY, USA, Jun. 2020, pp. 11–12.

[35] V. Terragni, P. Salza, and F. Ferrucci, "A container-based infrastructure for fuzzy-driven root causing of flaky tests," in *Proc. ACM/IEEE 42nd Int. Conf. Softw. Eng., New Ideas Emerg. Results*, New York, NY, USA, Jun. 2020, pp. 69–72.

[36] A. Shi, W. Lam, R. Oei, T. Xie, and D. Marinov, "IFixFlakies: A framework for automatically fixing order-dependent flaky tests," in *Proc. 27th ACM Joint Meeting Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng.*, New York, NY, USA, Aug. 2019, pp. 545–555.

[37] J. Malm, A. Causevic, B. Lisper, and S. Eldh, "Automated analysis of flakiness-mitigating delays," in *Proc. IEEE/ACM 1st Int. Conf. Autom. Softw. Test*, New York, NY, USA, Oct. 2020, pp. 81–84.

[38] A. Shi, A. Gyori, O. Legunsen, and D. Marinov, "Detecting assumptions on deterministic implementations of non-deterministic specifications," in *Proc. IEEE Int. Conf. Softw. Test., Verification Validation (ICST)*, Apr. 2016, pp. 80–90.

[39] D. Silva, L. Teixeira, and M. d'Amorim, "Shake it! Detecting flaky tests caused by concurrency with shaker," in *Proc. IEEE Int. Conf. Softw. Maintenance Evol. (ICSME)*, Sep. 2020, pp. 301–311.

[40] S. Dutta, A. Shi, R. Choudhary, Z. Zhang, A. Jain, and S. Misailovic, "Detecting flaky tests in probabilistic and machine learning applications," in *Proc. 29th ACM SIGSOFT Int. Symp. Softw. Test. Anal.*, New York, NY, USA, Jul. 2020, pp. 211–224.

[41] K. Herzig, "Let's assume we had to pay for testing," Keynote at AST, 2016. [Online]. Available: https://www.slideshare.net/kim.herzig/keynote-ast-2016

[42] J. Leskovec, A. Rajaraman, and J. D. Ullman, *Mining Massive Datasets*. New York, NY, USA: Cambridge Univ. Press, 2014.

[43] K. Beyer, J. Goldstein, R. Ramakrishnan, and U. Shaft, "When is 'nearest neighbor' meaningful?" in *Proc. Int. Conf. Database Theory*. Berlin, Germany: Springer, 1999, pp. 217–235.

[44] D. Achlioptas, "Database-friendly random projections: Johnson-Lindenstrauss with binary coins," *J. Comput. Syst. Sci.*, vol. 66, no. 4, pp. 671–687, Jun. 2003.

[45] P. Li, T. J. Hastie, and K. W. Church, "Very sparse random projections," in *Proc. 12th ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining*, 2006, pp. 287–296.

[46] W. B. Johnson and J. Lindenstrauss, "Extensions of lipschitz mappings into a Hilbert space," *Contemp. Math.*, vol. 26, pp. 189–206, May 1984.

[47] A. Magen, "Dimensionality reductions in ℓ_2 that preserve volumes and distance to affine spaces," *Discrete Comput. Geometry*, vol. 38, pp. 139–153, Jul. 2007.

[48] S. Elbaum, G. Rothermel, and J. Penix, "Techniques for improving regression testing in continuous integration development environments," in *Proc. 22nd ACM SIGSOFT Int. Symp. Found. Softw. Eng.*, New York, NY, USA, 2014, pp. 235–245.

[49] V. R. Basili, "Goal question metric paradigm," in *Encyclopedia of Software Engineering*, 1994, pp. 528–532.

[50] R. Kohavi, "A study of cross-validation and bootstrap for accuracy estimation and model selection," in *Proc. 14th Int. Joint Conf. Artif. Intell.*, vol. 2. San Francisco, CA, USA: Morgan Kaufmann, 1995, pp. 1137–1143.

[51] J. L. Bentley, "Multidimensional binary search trees used for associative searching," *Commun. ACM*, vol. 18, no. 9, pp. 509–517, Sep. 1975.

[52] S. M. Omohundro, *Five Balltree Construction Algorithms*. Berkeley, CA, USA: International Computer Science Institute Berkeley, 1989.

[53] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wessln, *Experimentation in Software Engineering*. Berlin, Germany: Springer, 2012.

ROBERTO VERDECCHIA is currently pursuing the double Ph.D. degree in computer science with the Vrije Universiteit Amsterdam, The Netherlands, and Gran Sasso Science Institute, L'Aquila, Italy.

He is currently a Research Associate with the Software and Sustainability Group (S2), Vrije Universiteit Amsterdam. His research interests include adoption of empirical methods to improve software development and system evolution, with particular emphasis in the fields of technical debt, software architecture, software testing, and software energy efficiency.

EMILIO CRUCIANI received the M.Sc. degree in engineering (computer science) from Sapienza University Rome, Rome, Italy, in 2016, and the Ph.D. degree in computer science from the Gran Sasso Science Institute, L'Aquila, Italy, in 2019.

From 2019 to 2020, he was a Postdoctoral Researcher with COATI Team, INRIA Sophia Antipolis Méditerranée, France, and he is currently a Postdoctoral Researcher with the Efficient Algorithms Group, University of Salzburg, Austria. His research interests include the analysis of stochastic processes on complex networks and the design and implementation of scalable and efficient algorithms for massive datasets.

BRENO MIRANDA received the master's degree in computer science from the Federal University of Pernambuco, Brazil, in 2011, and the Ph.D. degree in computer science from the University of Pisa, Italy, in 2016. He is currently an Assistant Professor with the Federal University of Pernambuco. His research interest includes software engineering, with particular focus in software testing.

ANTONIA BERTOLINO is currently a Research Director of the Italian National Research Council (CNR), Institute for Information Science and Technologies "Alessandro Faedo" (ISTI), Pisa, Italy. Her research covers a broad range of topics and techniques within software testing. She has published more than 200 papers in international journals, conferences, and workshops. She has participated to several collaborative projects, including more recently the European projects ElasTest, Learn Pad, and CHOReOS. She serves regularly in the Program Committee of top conferences in software engineering, such as ESEC-FSE, ICSE, Software Testing, and ISSTA, ICST. She currently serves as a Senior Associate Editor for the *Journal of Systems and Software* (Elsevier), and as an Associate Editor of *ACM Transactions on Software Engineering and Methodology*, *Empirical Software Engineering* (Springer), and *Journal of Software: Evolution and Process* (Wiley).