What is the Vocabulary of Flaky Tests?

Gustavo Pinto
Federal University of Pará
Belém, Brazil
gpinto@upfa.br

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

Supun Dissanayake
University of Adelaide
Adelaide, Australia
supun.dissanayake@adelaide.edu.au

Marcelo d'Amorim
Federal University of Pernambuco
Recife, Brazil
damorim@cin.ufpe.br

Christoph Treude
University of Adelaide
Adelaide, Australia
christoph.treude@adelaide.edu.au

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

ABSTRACT

Flaky tests are tests whose outcomes are non-deterministic. Despite the recent research activity on this topic, no effort has been made on understanding the vocabulary of flaky tests. This work proposes to automatically classify tests as flaky or not based on their vocabulary. Static classification of flaky tests is important, for example, to detect the introduction of flaky tests and to search for flaky tests after they are introduced in regression test suites.

We evaluated performance of various machine learning algorithms to solve this problem. We constructed a dataset of flaky and non-flaky tests by running every test case, in a set of 64k tests, 100 times (6.4 million test executions). We then used machine learning techniques on the resulting dataset to predict which tests are flaky from their source code. Based on features, such as counting stemmed tokens extracted from source code identifiers, we achieved an F-measure of 0.95 for the identification of flaky tests. The best prediction performance was obtained when using Random Forest and Support Vector Machines. In terms of the code identifiers that are most strongly associated with test flakiness, we noted that job, action, and services are commonly associated with flaky tests. Overall, our results provides initial yet strong evidence that static detection of flaky tests is effective.

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**.

KEYWORDS

test flakiness, regression testing, text classification

ACM Reference Format:
Gustavo Pinto, Breno Miranda, Supun Dissanayake, Marcelo d'Amorim, Christoph Treude, and Antonia Bertolino. 2020. What is the Vocabulary of Flaky Tests?. In *17th International Conference on Mining Software Repositories (MSR '20), October 5–6, 2020, Seoul, Republic of Korea.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3379597.3387482

MSR '20, October 5–6, 2020, Seoul, Republic of Korea

ACM ISBN 978-1-4503-7517-7/20/05…$15.00
https://doi.org/10.1145/3379597.3387482

1 INTRODUCTION

[[TODOS: 1) use dataset or data set consistently, 2) remove orphans, 3) squeeze figure 2, 4) use small fonts for tables (to save space)]]

Regression testing is an important practice in software development [7]. It aims to check that any code or configuration changes do not break existing functionality. Ideally, tests should be deterministic, i.e., their output should remain the same for the same environment and product configuration, and this is often assumed in academic research [11, 34]. Unfortunately, in practice, non-deterministic—or *flaky*—tests are common [11, 19, 20]. These are tests that may unpredictably pass or fail when rerun, even with no changes to the configuration under test.

In regression testing of large complex systems, developers may spend important resources in analyzing failures that are due to flaky tests and not to actual problems in production code, with concrete impact on productivity and costs. Practitioners got now used to rerun each newly observed failure several times, to ascertain that it is a genuine regression failure and not an intermittent one [16, 21]. However, this is a very inefficient way to deal with flakiness, and in recent years the software engineering community is observing an insurgence of research in approaches for preventing, identifying, and repairing flaky tests, e.g. [4, 13, 15, 17, 19, 26].

Notwithstanding, flaky tests remain deceitful. By manually analyzing the Apache Software Foundation (ASF) central commits repository, Luo et al. [19] aimed at identifying the prevalent causes of flakiness: they successfully identified a catalog of common causes explaining why a test is flaky. For instance, one of their findings was that "asynchronous wait" is the most common source of flakiness, responsible for 45% of the cases analyzed, and occurs when a test does not wait properly for the result of an asynchronous call.

Such types of study certainly help to understand the phenomenon, and also to reason on strategies to counteract it. For instance, the "asynchronous wait" problem can be fixed by introducing a method call with time delays (e.g., `Thread.sleep`). However, to really contrast flakiness, we need approaches that can *timely and efficiently recognize a flaky test, even well before it is committed in the test repository*. Preceding this study, we have analyzed different datasets of flaky tests [4, 8, 19, 23] and could observe that, as is the case for the "asynchronous wait" example, flaky tests seem to follow a set of syntactical patterns. Based on that, we conjecture that those test code patterns could be used to automatically recognize flaky tests using natural language processing (NLP) techniques.

To test this conjecture, we extracted identifiers, such as method names, from the code of test cases preventively labeled as flaky or non-flaky, and employed standard NLP techniques, including identifier splitting, stemming, and stop word removal, to turn these identifiers into tokens that could be used as input for text classification algorithms. We augmented these tokens with numerical features, such as the number of lines of code in the test case and the number of Java keywords, acting as proxies of code complexity, and ran five state-of-the-art classifiers on the resulting data set. The evaluation confirmed our conjecture, with our best classifier achieving an F_1-score of 0.95.

After analysing the impact of different features (e.g., identifier splitting) in our pipeline on the overall performance, we computed the *information gain* of each token, i.e., the usefulness of a token in distinguishing flaky tests from non-flaky tests. Tokens such as "job" and "table" showed particularly useful for this distinction, so that we can identify a sort of *vocabulary of flaky tests*, of which in the paper we provide a more detailed discussion.

Note that both the natural language processing and the prediction phase that form our approach can be carried out in a completely static way, i.e., without requiring any dynamic data such as coverage traces as is done in [4]. This is an important property of our approach, as collecting coverage information can be very costly, especially in Continuous Integration environments [12], whereas the overhead caused by our approach is expected to be negligible. Runtime cost consists of (1) extracting tokens from a test case (i.e., parsing), (2) post-processing the tokens (e.g., splitting words using their camel-case syntax), and (3) predicting the class of the exemplar using the previously-computed model.

In summary, the contributions of this work include:

(1) the first compilation of a vocabulary of flaky tests;
(2) a set of automated classifiers for test cases as flaky or non-flaky;
(3) performance evaluation of state-of-the-art classifiers over an existing data set of flaky tests.

Our contribution of a flaky test vocabulary and flakiness pattern classifiers can help: (1) to prevent the introduction of flaky tests by warning developers early, even while they are typing the test code (e.g., our approach could be embedded into the test code editor) and (2) to guide the identification of flaky tests that have been introduced in the test repository.

2 RELATED WORK

Our work is related with empirical studies of: *i)* test code bugs, *ii)* test smells and *iii)* flaky tests.

Test code bugs. A series of studies [30, 32] aims at characterizing causes and symptoms of buggy tests: these are problematic test cases that can fail raising a false alarm when in fact there is no indication of a bug in the application code. This paper focuses on test flakiness, which is one of several possible types of test code issues. Vahabzadeh et al. [30] mined the JIRA bug repository and the version control systems of ASF finding a set of 5,556 unique bug fixes exclusively affecting test code. They manually examined a sample of 499 test bugs and found that, among five identified major causes of false alarms, 21% were due to flaky tests, which they further classified into Asynchronous Wait, Race Condition and Concurrency Bugs. In contrast, to classify flaky tests we aim here at studying exclusively the test code, and not the fix changes, as they do. The authors of [32] developed a set of patterns that can help pinpoint problematic test code portions in JUnit test cases, and performed a validation study over a set of 12 open source projects. While their intent is similar to ours, we aim here at an automated lexical analysis of test cases. Recently, Tran et al. [27] studied test quality by surveying 19 practitioner's perceptions of test quality and conducting a mining study over the change history of 152 software projects, concluding that testers responsible for test execution are more concerned with comprehension of test cases rather than with their repeatability or performance.

Test smells. This research has been pioneered by van Deursen et al. [31] who identified a series of 11 different test smells, i.e., symptoms of poor design choices in test coding, and suggested a few refactoring guidelines. Recently, several extensive studies related to test smells have been conducted. Bavota et al. [2] and Tufano et al. [29] separately studied the test smell types defined in [31], which were detected through the application of simple comprehensive rules and then manual validation. Precisely, the study of Bavota et al. investigated their prevalence, concluding that up to 82% of 637 analyzed test classes contained at least one test smell, whereas Tufano et al. studied the life cycle of those smells, concluding that they are introduced since test creation (and not during test evolution), last for long surviving even thousands commits, and can be related to smells in production code. As flakiness may originate from test smells, such studies motivate our own study of code features in flaky tests. Indeed, in two subsequent studies [23, 24], Palomba and Zaidman analyze the relation between test smells and flakiness, and they observed that 75% of the flaky tests were due to presence of smells. We remain to investigate if a catalogued test smell suggests high-level features that a prediction model could use to further increase accuracy. A recent work towards such direction leverages information retrieval techniques [25], somewhat following a conjecture as the one we make here. It is also worth noting that a more comprehensive catalogue of test smells and a summary of guidelines and tools to deal with them are provided by Garousi et al. in a multivocal literature review [10].

Flaky tests. The first empirical study centered on flakiness is due to Luo et al. [19]. In this seminal work, they first filtered out from the complete commit history of the ASF central repository 1,129 commits including the keyword "flak" or "intermit", and then manually inspected all of them. As a result of their extensive work, they propose 10 categories of flakiness root causes, still widely referred, and summarize the most common strategies to repair them. Thorve et al. [26] conducted a similar study in Android apps, observing that some causes of Android tests flakiness are similar to those identified by Luo et al. [19], but also finding two new causes as Program Logic and UI. We are interested in identifying causes of flakiness as [19, 26], but we strive for automated and efficient detection of flakiness that could be applied, for example, to warn developers during evolution when they are about to add likely flaky tests. We remain to evaluate how our classifiers perform during evolution. We are particularly interested in understanding developers' reaction to the indication of potential flakiness produced by an IDE in

```
@Test
public void testCodingEmptySrcBuffer() throws Exception {
 final WritableByteChannelMock channel = new ritableByteChannelMock(64);
 final SessionOutputBuffer outbuf = new SessionOutputBufferImpl(1024, 128);
 final BasicHttpTransportMetrics metrics = new BasicHttpTransportMetrics();
 final IdentityEncoder encoder = new IdentityEncoder(channel, outbuf, metrics);
 encoder.write(CodecTestUtils.wrap("stuff"));
 final ByteBuffer empty = ByteBuffer.allocate(100);
 empty.flip();
 encoder.write(empty);
 encoder.write(null);
 encoder.complete();
 outbuf.flush(channel);
 final String s = channel.dump(StandardCharsets.US_ASCII);
 Assert.assertTrue(encoder.isCompleted());
 Assert.assertEquals("stuff", s);
}
```

⇓

```
pty src buffer codec test utils standard charsets
channel assert equals encoder byte buffer empty test
coding empty assert allocate flush outbuf metrics
dump complete wrap write flip stuff completed
```

Figure 1: A selected test case and its tokenized result.

contrast with the alternative approach that indicates flakiness in a report produced by Continuous Integration (CI) systems.

Our paper is also related with works that propose techniques to locate flaky tests. Bell et al. [4] and Lam et al. [17] proposed different techniques for detecting test flakiness *dynamically*, i.e., they require that test cases are executed (one or more times), aiming at optimizing the traditional approach used by practitioners of rerunning failed tests for a fixed number of times. Gambi et al. [9] focus on one specific cause of flakiness that is test dependency, which they propose to discover by flow analysis and iterative testing of possible dependencies. The works in [13, 15] aim instead to build a *static* predictor, as we also do here. The work in [13] develops a machine learning approach that mines association rules among individual test steps in tens of millions of false test alarms. In [15] a Bayesian network is instead constructed. In contrast, our work aims at developing a lightweight flakiness predictor that learns from test code of flaky and non-flaky tests. We are aware of one only recent approach that takes a similar standpoint as we do (i.e., [5]). However, here, we derive a more comprehensive set of predictors and build a vocabulary of tokens, which is out of their scope.

3 APPROACH

To understand the vocabulary of flaky tests, we extracted all identifiers from the test cases in our data set. We first localized the file declaring the test class and then processed that file to identify the flaky test case and corresponding identifiers. After obtaining the identifiers used in the test code, we split these identifiers using their camel-case syntax, and converted all resulting tokens to lower case. We removed stop words from the set of tokens for each test case. As a concrete example, consider the code snippet appearing at the top of Figure 1. This is a test case from the httpcore project[1]. The tokens extracted from the test appear at the bottom of the figure.

[1]https://tini.to/52IC

We observed that, in some cases, a part of an identifier after splitting (i.e., a token) seemed to be an indicator of flakiness (e.g., "services"), whereas, in other cases, the entire identifier was an indicator of flakiness (e.g., "getstatus"), but not its constituents on their own (e.g., "get", "status"). Therefore, we used both the split identifiers and the original identifiers (after lower-casing) as input for the text classification. In other words, the identifier "getStatus" would be represented using three features: "get", "status", and "getstatus". We evaluate the impact of this choice in our evaluation section.

In addition to the tokens obtained this way, we determined the length of each test case in terms of lines of code and the number of Java keywords contained in the test code, as a proxy for the code's complexity. Again, we separately evaluate the impact of these choices as part of answering our third research question.

We then used the pre-processed flaky and non-flaky test cases as input for machine learning algorithms. Each test case was represented using its features: the number of lines of code, the number of Java keywords, and for each token the information whether or not it contained this token. This approach creates one feature for each distinct token found in tests cases. Consequently, our data set includes a large number of features. Following previous work, we used attribute selection to remove features with low information gain: we used the same threshold of 0.02 as in previous work [28].

We evaluated the performance of five machine learning classifiers on our data set. We chose the same classifiers as used in previous work on text classification in the context of software engineering (e.g., [6, 28]): Random Forest, Decision Tree, Naive Bayes, Support Vector Machine, and Nearest Neighbour. For all algorithms, we relied on their implementation in the open source machine learning software Weka [33].

To evaluate the performance, we split our data set into 80% for training and 20% for testing. We choose to report the results based on this split rather than x-fold cross-validation since cross-validation would train a new model from scratch for each fold, thus resulting in several models rather than a single one. Note that we also ran our experiments with 10-fold cross-validation, with very similar (slightly improved) performance numbers. We report the standard metrics of precision (the number of correctly classified flaky tests divided by the total number of tests that are classified as flaky), recall (the number of correctly classified flaky tests divided by the total number of actual flaky tests in the test set), and F_1-score (the harmonic mean of precision and recall). We also report MCC (Matthews correlation coefficient) and AUC (area under the ROC curve). MCC measures the correlation between predicted classes (i.e., flaky vs. non-flaky) and ground truth, and AUC measures the area under the curve which visualises the trade-off between true positive rate and false positive rate. We focus our discussions on the F_1-score since we are more interested in correctly predicting flakiness rather than non-flakiness.

4 OBJECTS OF ANALYSIS

This section describes the datasets we used to train and test our prediction model. Machine learning algorithms use positive and negative examples for learning. In our setting, positive examples correspond to flaky test cases whereas negative examples correspond to likely non-flaky test cases. Indeed, the diagnosis of non-flakiness

Table 1: Projects and number of test cases analyzed.

project	description	GitHub ID	# tests	
			flaky	non-flaky
achilles	Java Object Mapper/Query DSL generator for Cassandra	doanduyhai/Achilles.git	67	8
alluxio	distributed storage system	Alluxio/alluxio.git	4	3022
ambari	manages and monitors Apache Hadoop clusters	apache/ambari.git	4	15
assertj-core	strongly-typed assertions for unit testing	joel-costigliola/assertj-core.git	-	13455
checkstyle	checks Java source code for adherence to standards	checkstyle/checkstyle.git	-	3169
commons-exec	executes external processes from within the JVM	apache/commons-exec.git	2	103
dropwizard	library for building production-ready RESTful web services	dropwizard/dropwizard.git	1	1641
hadoop	framework for distributed processing of large data sets	apache/hadoop.git	305	4475
handlebars	a tool for building semantic templates	jknack/handlebars.java.git	1	844
hbase	non-relational distributed database	apache/hbase.git	-	402
hector	interface to the Cassandra database	hector-client/hector.git	2	282
httpcore	low level HTTP transport components	apache/httpcore.git	2	1441
jackrabbit-oak	hierarchical content repository	apache/jackrabbit-oak.git	8	13172
jimfs	in-memory file system for Java 7+	google/jimfs.git	7	5833
logback	a logging framework for Java	qos-ch/logback.git	2	526
ninja	full stack web framework for Java	ninjaframework/ninja.git	18	1022
okhttp	manage HTTP sessions	square/okhttp.git	66	1663
oozie	workflow engine to manage Hadoop jobs	apache/oozie.git	856	729
orbit	framework for building distributed systems	orbit/orbit.git	8	-
oryx	framework for large scale machine learning	OryxProject/oryx.git	13	393
spring-boot	Java-based framework used to create micro services	spring-projects/spring-boot.git	15	8133
togglz	feature flags for the Java platform	togglz/togglz.git	11	441
undertow	non-blocking web server	undertow-io/undertow.git	-	607
wro4j	web resource optimizer	wro4j/wro4j.git	10	1146
zxing	barcode scanning library for Java	zxing/zxing.git	1	457
total	-	-	1,403	62,979

is an estimate—there is no guarantee a test is non-flaky with a given number of runs.

We based the construction of our dataset on the DeFlaker benchmark[2]. We took this decision based on the number of flaky test cases it reports, with over 5K flaky tests[3], which is, to the best of our knowledge, the largest dataset of flaky tests available today. In a nutshell, DeFlaker monitors the coverage of several Java projects. For each one of them, DeFlaker observes the latest code changes and marks as flaky any newly failing test that did not execute changed code. The expectation is that a test that used to pass and did not execute changed code should pass. As that was not the case, there must have been changes in the coverage profile caused by non-determinism.

In the following, we describe the methodology we used to construct the datasets. DeFlaker is focused on finding flaky test cases. Consequently, its benchmark does *not* list non-flaky tests, which are necessary for training a machine learning classifier. To circumvent this limitation, we re-executed the test suites of the projects from the DeFlaker benchmark for 100 times and flagged as (likely) non-flaky all test cases that had a consistent outcome across all executions, e.g., the test passes in all runs.

It is worth noting that, considering all test cases from all projects we analyzed, the number of non-flaky tests is much higher compared to the number of flaky tests and learning from imbalanced data is challenging. To mitigate this problem, we selected an equal number of non-flaky tests as that of flaky tests—original DeFlaker dataset—and selected each non-flaky test in a way that the median sizes (in number of lines of code) of flaky and non-flaky tests were nearly the same. More precisely, we proceeded as follows. Consider that the number of flaky test cases and their median sizes were, respectively, n and s. We randomly selected a test with size above s and then randomly selected a test with size below s. We repeated this selection process until selecting n distinct tests to complete the dataset. We empirically confirmed that the median sizes of the set of flaky and non-flaky test sets were very close.

Altogether, we considered 24 of the 25 DeFlaker projects, discarding one project—`orbit`. In the latter, we were unable to build the project, since the most recent version had build compilation errors. We also tried to navigate in the latest five revisions available in the version history, but we observed the same build problem. We then decided to discard this project from the rest of the analysis. All re-executions for non-flaky tests were made on the most recent revision of each of the 24 studied projects. Altogether, we ran 64k test cases over all the studied projects. The data produced in this work is available online at: https://github.com/damorimRG/msr4flakiness/

[2] www.deflaker.org/icsecomp/
[3] http://www.deflaker.org/wp-content/uploads/2019/11/historical_rerun_flaky_tests.csv

5 EVALUATION

Based on the approach described and the dataset curated, we pose the following research questions.

- **RQ1.** How prevalent and elusive are flaky tests?

Rationale. Prior work showed that flaky tests are common in regression test suites [4, 8, 11, 18, 19, 21, 23]. The goal of this research question is to confirm that phenomenon to justify the importance of statically classifying flaky tests, which is the central goal of this paper. To answer this question, we conducted an experiment where we ran the test suites of the 24 projects we selected (see Section 4) for 100 times on their latest revisions. We considered a test as flaky if there was a disagreement in the outcomes (i.e., pass, fail or error) across the hundred runs. For example, we consider as flaky a test that passes in all but one (or more) run(s). Given that most projects in that dataset are popular and that the teams had the chance to fix flaky tests originally reported in the DeFlaker paper, we considered those projects a good benchmark to check whether flaky tests are still present. Another dimension we wanted to analyze with this study is the degree of flakiness of each test. This is important to identify if there is an ideal number of reruns that one could use to find flakiness. If that number is sufficiently small then rerunning test suites may be considered a practical approach to detect flakiness.

- **RQ2.** How accurately can we predict test flakiness based on source code identifiers in the test cases?

Rationale. Being able to predict test flakiness based on source code identifiers would enable us to notify developers of flaky tests without having to run these tests. This would be particularly important since we conjecture that flaky tests might take a while to run because they might rely on time-intensive actions such as connecting to external services, for example, and since it is impossible to determine flakiness based on a single run or even several runs (see Section 5.1). Of course, such a recommender system, which warns developers when they are about to introduce a flaky test, can only be useful if the precision of the approach is high—developers would not appreciate false positives, i.e., being warned about flaky tests which are not actually flaky. Therefore, in answering our second research question, we seek to evaluate the performance of classifiers to predict test flakiness without running the tests, i.e., based on the source code identifiers.

- **RQ3.** What value do different features add to the classifier?

Rationale. Understanding what features affect the performance of the classifier will help inform future work in areas where further performance gains might be possible. We employ standard pre-processing steps, such as stemming and stop word removal, in our approach, but also want to evaluate to what extent these steps affect the performance of the classifiers. In particular, when converting source code identifiers into numeric features amenable to traditional machine learning algorithms, we need to make several design choices, such as deciding whether to split identifiers. We also want to evaluate the impact of these choices to guarantee the best possible performance of the classifier.

Table 2: Number of flaky tests per project. #PF (resp., #PE) denotes number of Pass and Fail (resp., Pass and Error) tests.

project	SHA	# test cases	# flaky tests (%)	#PF	#PE
alluxio	260533d	3,034	12 (0.40)	1	11
hector	a302e68	322	40 (12.4)	3	37
jackrabbit-oak	226e216	13,193	2 (0.02)	1	1
okhttp	6661e14	1,682	19 (1.20)	19	0
undertow	b6bd4d2	609	2 (0.33)	1	1
wro4j	d2a3de7	1,158	11 (0.95)	0	11
—	—	19,998	86 (—)	25	61

- **RQ4.** Which test code identifiers are most strongly associated with test flakiness?

Rationale. Cataloguing the test code identifiers that are strongly associated with test flakiness can inform software developers of particular aspects of developing software that are likely to lead to flaky tests. Based on this information, developers might be helped to prevent test flakiness, or at least be aware and pay extra care to areas that are likely to be associated with flaky tests. Such information could, for example, be useful when conducting code review and when debugging test failures.

5.1 Answering RQ1: How prevalent and elusive are flaky tests?

Table 2 shows the results for this first research question. The table shows the project's name (column "project"), their revision (column "SHA"), the total number of tests (column "# test cases"), the number of flaky tests found in that revision of that project (column "# flaky tests (%)"), and the breakdown of kind of flakiness: PF indicates a mix of pass and fail runs and PE indicates a mix of pass and error runs. We did not find other combination of test outcomes in these configurations. The table only includes projects with at least one flaky test detected. This result indicates that flakiness is indeed a problem affecting 25% (=6/24) of the projects analyzed. Overall, we found a total of 86 flaky tests by rerunning test cases. The project alluxio is a virtual distributed storage system, hector is a high-level Java client interface to the Cassandra distributed database, jackrabbit-oak is an efficient implementation of a hierarchical content repository for use in web sites and content-management systems, okhttp is a library to efficiently manage HTTP sessions, undertow is a high-performance non-blocking web server implementation, and wro4j is a library to optimise web page loading time. Note that every project involves IO, for example, they refer to the file system or the network.

Another interesting finding of this table is the low number of flaky tests. Some reasons that may justify this result: First, running 100 times might not be enough to find a good number of flaky tests. However, it is not in the scope of this paper to empirically evaluate the ideal number of reruns to find flaky tests. Second, we used the most recent version of the DeFlaker dataset. Maintainers of these projects could have fixed the known flaky tests. Third, our focus on unit tests might cap the total number of flaky tests that could be observed.

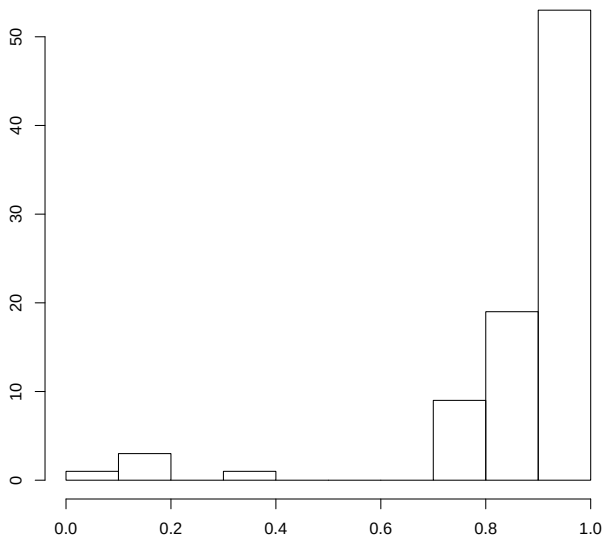

Figure 2: Histogram of probability of a flaky test to pass.[[Marcelo: if not too much trouble, can someone reduce aspect ratio (squeeze y-axis) of this figure as to avoid waste of vertical space?]]

Figure 2 shows the histogram for the probability of a flaky test to pass, with data aggregated across all projects. The x-axis shows probability intervals (e.g., [0, 10%[, [10%, 20%[, etc.) whereas the y-axis shows the number of flaky tests that fall in that interval.

The histogram shows that the majority of the cases we found to be flaky, around 70% (61 out of 86), passed in more than 90% of the executions. For example, for 47 flaky tests (55%), the test passed 99 times (out of 100 repetitions) and produced a different result in only one case. This result may indicate that more executions might be needed to accurately identify flaky tests.

The histogram also shows that there are rare cases where the probability of a flaky test to pass is low–only one flaky test passed in less than 10% of the executions. For this case, the strategy adopted by Continuous Integration (CI) systems to rerun the test for a small number of times would unlikely identify the cause of failure as flakiness. The probability of subsequent failures after the first test execution fails is relatively high. Assuming for example that the framework reruns a test three other times, after a failure, the probability of flakiness going undetected would be 66% (=0.9^4), i.e., the probability of four failures in a row.

Results indicate that flakiness is a relatively common problem in IO-related projects. Furthermore, detecting flakiness with test reruns is challenging.

5.2 Answering RQ2: How accurately can we predict test flakiness based on source code identifiers in the test cases?

Table 3 shows the performance of five machine learning algorithms on our dataset in terms of standard metrics used in the literature, namely: precision, recall, F_1-score, MCC (Matthews correlation coefficient), and AUC (area under the ROC curve). Numbers in bold highlight the algorithm that performed best for a given metric.

Table 3: Classifier performance

algorithm	precision	recall	F_1	MCC	AUC
Random Forest	**0.99**	0.91	**0.95**	**0.90**	**0.98**
Decision Tree	0.89	0.88	0.89	0.77	0.91
Naive Bayes	0.93	0.80	0.86	0.74	0.93
Support Vector	0.93	**0.92**	0.93	0.85	0.93
Nearest Neighbour	0.97	0.88	0.92	0.85	0.93

All classifiers achieved very good performance in distinguishing flaky test cases from non-flaky test cases. While Random Forest achieved the best precision (0.99), the Support Vector Machine classifier slightly outperformed Random Forest in terms of recall (0.92). Overall, in terms of F_1-score, Random Forest achieved the best performance, but all classifiers achieved an F_1-score of at least 0.85. Results are consistent when considering Matthews correlation coefficient and area under the ROC curve. In both cases, the Random Forest classifier achieves the best performance, with values of 0.90 and 0.98, respectively.

As is common when using automated classifiers, we attempted parameter tuning to see if it would impact the classifier performance. In this case, we changed the 'number of trees' parameter of the Random Forest algorithm from its default setting in Weka of 100. Increasing the number of trees had no impact on the F_1-score (we tried values of 500 and 1,000) while reducing the number of trees led to a decrease in F_1-score to 0.91 for the values of 5 and 10. Reducing the number of trees to 50 had no impact on the F_1-score.

All classifiers performed very well on our dataset. Overall, Random Forest was the classifier that performed best.

5.3 Answering RQ3: What value do different features add to the classifier?

In this section, we investigate the impact of the different features used in our classifiers on their performance. We focus the investigation on the two best-performing classifiers identified in the previous section: Random Forest (best precision and F_1-score) and Support Vector Machine (best recall).

Tables 4a and 4b compare the performance of these two classifiers to the performance of the same classifier without a particular feature, including features of the text classification algorithm (e.g., stemming, stop word removal, etc.) and features describing the data (e.g., number of lines of code, contains identifier "status", etc.).

Table 4: Performance without features

(a) Random Forest

features	precision	recall	F_1	MCC	AUC
All Features	0.99	0.91	0.95	0.90	0.98
No Stemming	0.99	0.91	0.95	0.90	0.98
No Stop W. Removal	0.99	0.91	0.95	0.90	0.98
No Lowercasing	0.98	0.91	0.94	0.89	0.98
No Identifier Split.	0.98	0.89	0.94	0.88	0.98
Only Split Identif.	0.99	0.92	0.95	0.90	0.98
No Lines of Code	0.99	0.91	0.95	0.90	0.99
No Java Keywords	0.99	0.90	0.94	0.89	0.98
No Identifiers	0.76	0.82	0.79	0.56	0.85

(b) Support Vector

features	precision	recall	F_1	MCC	AUC
All Features	0.93	0.92	0.93	0.85	0.93
No Stemming	0.93	0.92	0.93	0.85	0.93
No Stop W. Removal	0.93	0.92	0.93	0.85	0.93
No Lowercasing	0.91	0.93	0.92	0.84	0.92
No Identifier Split.	0.91	0.88	0.89	0.79	0.90
Only Split Identif.	0.93	0.92	0.93	0.85	0.93
No Lines of Code	0.93	0.92	0.93	0.85	0.93
No Java Keywords	0.93	0.92	0.93	0.85	0.93
No Identifiers	0.64	0.87	0.74	0.40	0.68

For the Random Forest classifier (Table 4a), not all features in our pipeline had a visible impact on the results: running the same pipeline, but without stemming, without stop word removal or without including the LOC metric had no impact on the F_1-score, for example, and it also made no difference whether we considered only split identifiers as tokens (e.g., turning `getId` into two features `get` and `id` instead of three features `get`, `id`, and `getid`). Lowercasing had a negligible impact (without it, the F_1-score would drop from 0.95 to 0.94), similar to not including Java keywords or not splitting identifiers by camel case.

The only large impact was observed when we only included Java keywords as tokens, but not identifier names. In this case, the performance would drop from an F_1-score of 0.95 to 0.79.

As Table 4b shows, the results for the Support Vector Machine classifier are similar: the F_1-score was not affected by stemming, stop word removal, the LOC metric, and Java keywords, while the effect of lowercasing was negligible. Not splitting identifiers reduced the F_1-score from 0.93 to 0.89 and not considering identifiers at all reduced it to 0.74.

> While the impact of some pre-processing steps is negligible, identifier splitting has a positive impact on the classifier performance.

Table 5: Top 20 features by Information Gain

feature	inf. gain	flaky		non-flaky	
		#tests	#projects	#tests	#projects
job	0.2053	524	(2)	4	(1)
table	0.1449	406	(4)	8	(2)
id	0.1419	522	(9)	52	(4)
action	0.1366	387	(3)	8	(2)
oozie	0.1360	274	(1)	0	(0)
services	0.1310	371	(2)	7	(1)
coord	0.1192	307	(1)	0	(0)
getid	0.1077	287	(4)	1	(1)
coordinator	0.1070	258	(1)	0	(0)
xml	0.1062	147	(2)	6	(2)
LOC (metric)	0.0978	-	-	-	-
workflow	0.0914	207	(1)	0	(0)
getstatus	0.0885	246	(2)	2	(2)
`throws` (Java)	0.0874	3	(3)	7	(2)
record	0.0845	296	(2)	18	(1)
jpa	0.0781	207	(2)	0	(0)
jpaservice	0.0753	200	(1)	0	(0)
service	0.0733	367	(4)	67	(3)
wf	0.0721	192	(1)	0	(0)
coordinatorjob	0.0689	184	(1)	0	(0)

5.4 Answering RQ4: Which test code identifiers are most strongly associated with test flakiness?

Table 5 shows the 20 features with the highest information gain along with their frequency in flaky and non-flaky test cases. The table also shows in how many different projects each of these features appeared. We discuss the most prominent features in more detail in the following paragraphs.

The feature with the highest information gain is that associated with the token "job", i.e., the feature "is the token job included in the test case?". This feature appeared in 524 different flaky tests in our data set, distributed across 2 projects (Hadoop and Oozie), but only in 4 different non-flaky tests, all from the same project (Hadoop). An example of a flaky test which contains the token "job" more than ten times is `testFailAbortDoesntHang` in the Hadoop project. Figure 3 shows the code for this test. The test creates and aborts jobs within a ten second time budget—the timeout is likely the reason that the test case sometimes fails and sometimes does not. Several of the other test cases associated with flakiness and the token job are about killing a job, e.g., `testKill` and `testCoordKillSuccess1`.

The feature with the second highest information gain is that associated with the token "table", appearing in 406 flaky tests across four projects (Achilles, Hadoop, Oozie, and OkHttp) and in eight non-flaky tests across two projects (Hadoop and HttpCore). An example is the test `testTableCreateAndDeletePB` from Hadoop which contains the token more than ten times. Figure 4 shows the code for this test case. The code suggests that the need to wait for a table to come online after a call to method `enableTable` might be the reason for flakiness. Other flaky test cases containing the token "table" are similar, e.g., `testDisableAndEnableTable` and `testWritesWhileScanning`. Connecting to tables and/or databases appears to be a source for flakiness.

```
@Test(timeout = 10000)
public void testFailAbortDoesntHang() throws IOException {
 Configuration conf = new Configuration();
 conf.set(MRJobConfig.MR_AM_STAGING_DIR, stagingDir);
 conf.set(MRJobConfig.MR_AM_COMMITTER_CANCEL_TIMEOUT_MS, "1000");
 DrainDispatcher dispatcher = new DrainDispatcher();
 dispatcher.init(conf);
 dispatcher.start();
 OutputCommitter committer = Mockito.mock(OutputCommitter.class);
 CommitterEventHandler commitHandler =
   createCommitterEventHandler(dispatcher, committer);
 commitHandler.init(conf);
 commitHandler.start();
 // Job has only 1 mapper task. No reducers
 conf.setInt(MRJobConfig.NUM_REDUCES, 0);
 conf.setInt(MRJobConfig.MAP_MAX_ATTEMPTS, 1);
 JobImpl job = createRunningStubbedJob(conf, dispatcher, 1, null);
 // Fail. finish all the tasks. This should land the JobImpl directly in the
       FAIL_ABORT state
 for (Task t : job.tasks.values()) {
   TaskImpl task = (TaskImpl) t;
   task.handle(new TaskEvent(task.getID(), TaskEventType.T_SCHEDULE));
   for (TaskAttempt ta : task.getAttempts().values()) {
     task.handle(new TaskTAttemptEvent(ta.getID(), TaskEventType.
           T_ATTEMPT_FAILED));
   }
 }
 assertJobState(job, JobStateInternal.FAIL_ABORT);
 dispatcher.await();
 // Verify abortJob is called once and the job failed
 Mockito.verify(committer, Mockito.timeout(2000).times(1)).abortJob((JobContext
       ) Mockito.any(), (State) Mockito.any());
 assertJobState(job, JobStateInternal.FAILED);
 dispatcher.stop();
}
```

Figure 3: Code for test `TestJobImpl.testFailAbortDoesntHang` from project Hadoop with prolific use of term "job".

```
public void testTableCreateAndDeletePB() throws IOException, JAXBException {
 String schemaPath = "/" + TABLE2 + "/schema";
 TableSchemaModel model;
 Response response;
 assertFalse(admin.tableExists(TABLE2));
 // create the table
 model = TestTableSchemaModel.buildTestModel(TABLE2);
 TestTableSchemaModel.checkModel(model, TABLE2);
 response = client.put(schemaPath, Constants.MIMETYPE_PROTOBUF, model.
       createProtobufOutput());
 assertEquals(response.getCode(), 201);
 // make sure HBase concurs, and wait for the table to come online
 admin.enableTable(TABLE2);
 // retrieve the schema and validate it
 response = client.get(schemaPath, Constants.MIMETYPE_PROTOBUF);
 assertEquals(response.getCode(), 200);
 model = new TableSchemaModel();
 model.getObjectFromMessage(response.getBody());
 TestTableSchemaModel.checkModel(model, TABLE2);
 // delete the table
 client.delete(schemaPath);
 // make sure HBase concurs
 assertFalse(admin.tableExists(TABLE2));
}
```

Figure 4: Code for test method `testTableCreateAndDeletePB` from class `TestSchemaResource`, project Hadoop, with high usage of term "table".

"Id" is a common token in many software development projects and it is the third most useful token for distinguishing flaky test cases from non-flaky test cases in our data set. It appears in 522 flaky test cases across nine projects (Cloudera Oryx, Orbit, OkHttp, Achilles, Ambari, Hadoop, Jackrabbit Oak, Oozie, and Togglz), and in 52 non-flaky test cases across four projects (ZXing, Ninja, Hadoop, HttpCore). In addition, the token getid is the features with the eighth-highest information gain, appearing in 287 flaky test cases across four projects (Cloudera Oryx, Hadoop, Jackrabbit Oak, Oozie) and only in a single non-flaky test case in Hadoop. As an example, the test method `testUpdatedNodes` in Haddop contains the token id more than ten times. In this method, id is used to refer to different objects: jobs, attempts, applications, and nodes. Much like the example described in the context of the token job, in this case, the test method relies on jobs being completed elsewhere, which might contribute to its flakiness.

```
public void testActionExecutor() throws Exception {
 ActionExecutor.enableInit();
 ActionExecutor.resetInitInfo();
 ActionExecutor ae = new MyActionExecutor();
 ae.initActionType();
 ActionExecutor.disableInit();
 ae.start(null, null);
 ae = new MyActionExecutor(1, 2);
 ae.check(null, null);
 Exception cause = new IOException();
 try {
  throw ae.convertException(cause);
 } catch (ActionExecutorException ex) {
  assertEquals(cause, ex.getCause());
  assertEquals(ActionExecutorException.ErrorType.TRANSIENT, ex.getErrorType());
  assertEquals("IO", ex.getErrorCode());
 } catch (Exception ex) {
  fail();
 }
 ...
 // omitted for space
}
```

Figure 5: Code for test `TestActionExecutor.testActionExecutor` from Oozie, with high usage of term "action".

The token "action" occurred in 387 different flaky test cases across three projects (Ambari, Hadoop, and Oozie) and in eight different non-flaky test cases across two projects (Hadoop and Logback). An example is the test method `testActionExecutor` in Oozie which contains the token action four times. Figure 5 shows the code for this test method, which attempts to execute an action through a remote method invocation (RMI), likely the source of flakiness, e.g., because of timing issues in asynchronous calls or the remote object not listening to synchronous calls.

Table 5 shows further tokens associated with flakiness. Interestingly, we did not find a single token in the top 20 that was more strongly associated with non-flakiness. With the exception of the Java keyword `throws`, for all features shown in the table, a higher value indicates a higher likelihood of flakiness. In contrast, for the Java keyword `throws`, a lower value indicates a higher likelihood for flakiness. We conjecture that proper exception handling as indicated through the Java keyword `throws` can help avoid test flakiness.

The vocabulary associated with flaky tests contains words such as job, table, and action, many of which are associated with executing tasks remotely and/or using an event queue.

6 DISCUSSION

6.1 Threats to Validity

Threats to the construct validity are related to the appropriateness of the evaluation metrics we used. We report precision, recall, F_1-score, MCC (Matthews correlation coefficient), and AUC (area under the ROC curve), which have been used in many software engineering tasks that require classification (e.g., [14]). Our conclusions are mostly based on precision and F_1-score since these two metrics capture the usefulness of a recommender system that could warn developers when they are about to introduce a flaky test.

Threats to the internal validity compromise our confidence in establishing a relationship between the independent and dependent variables. While we have evidence for the flakiness of the test cases that we consider as flaky, it is possible that some of the test cases that we consider as non-flaky are actually flaky. When performing our first experiment (running the test cases of 24 Java projects 100 times to find flaky tests), we noticed that 55% of the test cases passed 99 times, and failed just once. This result suggests that the strategy of rerunning tests several times to detect flakiness could miss cases of flakiness as tests could have been insufficiently executed. Consequently, considering our experiment, in particular, we could have detected more cases of flaky tests if we executed each test more times. This threat can be mitigated only by performing additional, more extensive, experiments. Another internal validity threat may be related to the parameters chosen for applying the algorithms investigated. This threat was mitigated by tuning the parameters with values that are standard in this kind of work.

Threats to external validity relate to the ability to generalize our results. We cannot claim generalization of our results beyond the particular test cases studied. In particular, our findings are intrinsically limited by projects studied, as well as their domains. Although the studied projects are mostly written in Java, we do not expect major differences in the results if another object-oriented programming language is used instead, since some keywords may be shared among them. Nevertheless, future work will have to investigate to what extent our findings generalize to software written in other programming languages and software of different application domains. We are also eager to validate our results on a much larger selection of flaky tests. Curiously, we noticed in the experiment of RQ1 (Section 5.1) that all projects manifesting flakiness are IO-intensive. We should revisit that hypothesis by looking for IO and non-IO-intensive projects in the future. The validity of that hypothesis would enable us to find other projects with flaky tests as to augment our data set. Due to the limited size of the data set, we did not attempt within-project classification. Future work will investigate the extent to which this is possible as well as the differences between classifiers trained on different projects. Moreover, one might wonder why can the words used in test cases predict flaky tests so well. Table 5, which shows for several keywords how often they appear in flaky and non-flaky tests, might help answer this question. For some of these keywords, the differences are extreme, e.g., "job" occurs in 524 flaky tests and in 4 non-flaky tests. A classifier guessing that all tests containing the term "job" are flaky would by definition already achieve a precision of 99.2% (524/528). With similar ratios for other keywords and the power of Random Forest and Support Vector Machine, respectively, these differences translate into an excellent performance of the classifiers in terms of precision.

6.2 Lessons Learned

We elaborate in the following the main lessons we learned from this work.

On the observed results. We used the same set of machine learning algorithms for the classification of test cases that have been used in many previous studies (e.g., [6]). The finding that our best performance was achieved by Random Forest (all metrics but recall) and Support Vector Machines (recall) is in line with previous text classification studies in software engineering (e.g., [1, 28]). The observed performance of the classifiers was very good (F_1-measure = 0.95), which is a better result than obtained in most other text classification problems for software engineering. This suggests that source code identifiers carry much of the information needed to determine whether a test case is flaky.

On the efficiency of the approach. In this work the efficiency of our approach was not measured experimentally. We will conduct such a study as part of our future work and we expect it will confirm our expectation that the overhead of our approach is minimum. Being a completely static approach, most of the steps are completed in negligible time. Conceptually, the cost of predicting whether a test is flaky or not consists of (1) extracting tokens from a test case (i.e., parsing), (2) post-processing the tokens (e.g., splitting words using their camel-case syntax), and (3) predicting the class of the exemplar using the previously-computed model. The cost of building the model is also relatively very low. For the 2,250 test cases in our training set (80% of 1,403 flaky and 1,403 non-flaky tests), preparing the test cases was instantaneous (remove stop words, collect tokens, split identifiers, etc.) and training the Random Forest classifier with 100 iterations took 11.81 seconds. When new test cases are added to the project, or when existing ones are updated or removed from the test suite, our current approach would require re-training the model. For larger sets of training data, in future work, we will explore the use of models that can easily be updated.

Feature Selection. Our results showed that some of the pre-processing steps such as stemming and stop word removal only had negligible impact on the classifier performance. However, the way that source code identifiers were split affected the performance. In our current approach, each distinct token (after pre-processing) is considered as a separate feature and we employ feature selection based on information gain to reduce the number of features in the classifier. While machine learning algorithms implicitly take into account relationships between these tokens, future work could make this more explicit. For example, we found anecdotal evidence for tokens such as "job" and "action" to co-occur in test cases. Future work could explicitly consider investigating the importance of such

combination for the prediction of flakiness. Interestingly, we found split identifiers (e.g., "id", "job") as well as complete identifiers (e.g., "getid", "coordinatorjob") among the features with the highest information gain, suggesting that there is no clear rule as to whether or not split identifiers are more useful for classification than complete ones. We expect similar conclusions to apply to scenarios of co-occurrence of source code identifiers.

On the rerunning strategy. Rerunning failing test cases for identifying flaky tests can be very costly. During the process of building our data set we noticed that many of the projects (e.g., jackrabbit-oak and hadoop) take hours to run the whole test suite. For instance, for hadoop, one execution of a test suite takes about 94 minutes on an Intel Xeon machine (ES-2660) with 40 processors (2.20GHz) with 251GB of main memory. Repeating the execution many times would not be doable in many industrial environments. Even big companies with enormous computational resources (e.g., Google) cannot afford rerunning every failing test on every commit [22]. That said, the rerunning strategy remains an interesting alternative-unless we can derive approaches that are effective in identifying flaky tests while remaining efficient. Our approach is an attempt of providing such an effective and efficient solution for the flakiness problem. Note that static detection of flakiness could be used to reduce cost of rerunning test cases on failures. For example, a continuous integration system could be triggered to rerun a test only when that test becomes suspicious as per the output of the prediction models proposed in this paper.

6.3 Implications

This work has implications to both research and practice.

Research. In this work we noticed that an arbitrary number of re-executions might not be the best solution for finding flaky tests. Researchers could take advantage of this finding and explore other approaches, such as experimenting with a dynamic threshold to more sophisticated techniques such as finding an optimal threshold using search based optimization. Moreover, researchers could use the vocabulary of flaky tests and conduct additional experiments with them. For instance, researchers could investigate the proportion of builds failing in continuous integration systems that happen to have any of the features observed in our work. Researchers could also propose other machine learning algorithms that could be more suitable to work with flaky test data.

Practice. Practitioners could also take advantage of our findings. When learning from the top 20 features, developers could keep one eye open when writing their tests and try to avoid such terms (and eventual related terms). Similarly, code reviewers could easily spot such terms and suggest developers to propose another solution. Testing framework maintainers could also take advantage of this finding by proposing mocking frameworks that could introduce (or even recommend) mock strategies tailored to deal with flaky scenarios. Still, tool builders could warn developers when suspicious flaky-terms are used in the software development process.

7 CONCLUSION

Flaky tests are test cases that sometimes pass and sometimes fail, without any obvious change in the test code or in its execution environment. Unfortunately, the non-deterministic behaviour of flaky tests could severely decrease the value of an automated regression suite. For instance, when dealing with flaky tests, developers may not trust the outcome of these tests and ultimately may start ignoring if a test failure is due to a real bug or its non-deterministic behaviour. In the last few years, research on test flakiness has gained significant momentum. Prior work focused on characterizing what is a flaky or identifying the root cause of flaky tests. However, little effort has been placed on how to efficiently recognize a flaky test. This paper focuses on the question of whether there are programming identifiers (e.g., method and variable names) that could be used to automatically recognize flaky tests. More precisely, the paper proposes to answer the question: Is there a programming vocabulary that could distinguish flaky tests from their non-flaky relatives?

To answer this question, we started by extracting test cases from a well-known dataset of flaky tests [3]. Since we needed to have flaky and non-flaky tests and the data set only provided flaky data, we decided to rerun the Java projects studied in this data set, but now keeping an eye open for finding flaky tests. We then ran 100 times the 64k test cases of the 24 studied Java projects. We flag a test as flaky if there was disagreement in the test outcomes. After the identification of flaky tests, we extract all identifiers from the test cases using traditional tokenization procedures. Finally, the pre-processed flaky and non-flaky test cases were used as input to five machine learning algorithms.

Based on this data and approach, we could observe several interesting findings. First, we were able to find six projects with flaky tests, and a total of 86 flaky tests. More interestingly, however, is the fact that 55% of these flaky tests failed just once, meaning that the 100 threshold might have limited the observation of flaky tests (i.e., it is likely that we could find more flaky tests if we run many other executions). Second, we observed that the five machine learning algorithms used had good performance in distinguishing flaky from non-flaky tests. In particular, Random Forest had the best precision (0.99), while Support Vector Machine slightly outperformed Random Forest in terms of recall (0.92 vs 0.91). Third, in terms of the features used in the classifiers for improving performance, we noticed that, for both Random Forest and Support Vector Machine, perhaps surprisingly, most of the features in the classifier did not have a visible impact on the results. Finally, regarding the vocabulary of flaky tests, we noticed that words such as job, table, or action (which are often associated with remote work) are among the features with the highest information again.

Future work. We have plans for several other works along the lines of this work. First, we plan to create tools that could help developers in identifying flaky tests. Initially, these tools could receive as input the features we found with the highest information gain. These tools could also allow developers to confirm whether a test is flaky or not, and based on this decision, these tools could interactively improve their own dictionary of flaky-related words. We also plan to study how combination of different features could help improve accuracy of our prediction models. Although many machine learning algorithms analyze such combinations internally, proposing combination features explicitly may be helpful.

We also plan to do a qualitative study (using not only coding techniques, but also instrumentation and debugging techniques) over the sample of flaky tests we found, in order to properly reason about the flakiness. Still, we also have plans to reproduce the DeFlaker work [3]. In this paper, we found a very small number of flaky tests, when compared to the DeFlaker work. A careful reproduction would enable us to understand why we observed so few flaky tests and, consequently, how we could improve our approach for finding flaky tests.

Acknowledgements. This research was partially funded by INES 2.0, FACEPE grants PRONEX APQ 0388-1.03/14 and APQ-0399-1.03/17, CAPES grant 88887.136410/2017-00, and CNPq grant 465614/2014-0. It was also supported by FAPESPA, UFPA, and by a gift from a Facebook Research 2019 TAV (Testing and Verification) award.

REFERENCES

[1] Syed Nadeem Ahsan, Javed Ferzund, and Franz Wotawa. 2009. Automatic software bug triage system (bts) based on latent semantic indexing and support vector machine. In *2009 Fourth International Conference on Software Engineering Advances*. IEEE, 216–221.

[2] Gabriele Bavota, Abdallah Qusef, Rocco Oliveto, Andrea De Lucia, and David Binkley. 2012. An empirical analysis of the distribution of unit test smells and their impact on software maintenance. In *2012 28th IEEE International Conference on Software Maintenance (ICSM)*. IEEE, 56–65.

[3] Jonathan Bell, Owolabi Legunsen, Michael Hilton Lamyaa Eloussi, Tifany Yung, and Darko Marinov. 2018. Deflaker Dataset. http://www.deflaker.org/icsecomp/.

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

[5] Antonia Bertolino, Emilio Cruciani, Breno Miranda, and Roberto Verdecchia. 2020. Know Your Neighbor: Fast Static Prediction of Test Flakiness. https://doi.org/10.5281/zenodo.3610610

[6] Lucas BL de Souza, Eduardo C Campos, and Marcelo de A Maia. 2014. Ranking crowd knowledge to assist software development. In *Proceedings of the International Conference on Program Comprehension*. ACM, 72–82.

[7] Hyunsook Do. 2016. Recent Advances in Regression Testing Techniques. *Advances in Computers* 103 (2016), 53–77. https://doi.org/10.1016/bs.adcom.2016.04.004

[8] Moritz Eck, Fabio Palomba, Marco Castelluccio, and Alberto Bacchelli. 2019. Understanding Flaky Tests: The Developer's Perspective *(ESEC/FSE 2019)*. Association for Computing Machinery, New York, NY, USA, 830–840. https://doi.org/10.1145/3338906.3338945

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

[10] Vahid Garousi, Baris Kucuk, and Michael Felderer. 2018. What we know about smells in software test code. *IEEE Software* 36, 3 (2018), 61–73.

[11] Mark Harman and Peter W. O'Hearn. 2018. From Start-ups to Scale-ups: Opportunities and Open Problems for Static and Dynamic Program Analysis. In *Proc. SCAM'18*.

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

[13] Kim Herzig and Nachiappan Nagappan. 2015. Empirically Detecting False Test Alarms Using Association Rules *(ICSE '15)*. 39–48.

[14] Sunghun Kim, E James Whitehead Jr, and Yi Zhang. 2008. Classifying software changes: Clean or buggy? *IEEE Transactions on Software Engineering* 34, 2 (2008), 181–196.

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

[16] Wing Lam, Patrice Godefroid, Suman Nath, Anirudh Santhiar, and Suresh Thummalapenta. 2019. Root Causing Flaky Tests in a Large-scale Industrial Setting *(ISSTA 2019)*. ACM, New York, NY, USA, 101–111. https://doi.org/10.1145/3293882.3330570

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

[18] Jeff Listfield. 2017. Where do our flaky tests come from? https://testing.googleblog.com/2016/05/flaky-tests-at-google-and-how-we.html.

[19] Qingzhou Luo, Farah Hariri, Lamyaa Eloussi, and Darko Marinov. 2014. An Empirical Analysis of Flaky Tests. In *Proc. FSE'14*.

[20] John Micco. 2016. Flaky tests at Google and how we mitigate them. https://testing.googleblog.com/2016/05/flaky-tests-at-google-and-how-we.html. Accessed: 2020-01-15.

[21] John Micco. 2016. Flaky Tests at Google and How We Mitigate Them. https://testing.googleblog.com/2017/04/where-do-our-flaky-tests-come-from.html.

[22] John Micco. 2017. The State of Continuous Integration Testing @Google.

[23] F. Palomba and A. Zaidman. 2017. Does Refactoring of Test Smells Induce Fixing Flaky Tests?. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 1–12. https://doi.org/10.1109/ICSME.2017.12

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

[25] Fabio Palomba, Andy Zaidman, and Andrea De Lucia. 2018. Automatic test smell detection using information retrieval techniques. In *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 311–322.

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

[27] Huynh Khanh Vi Tran, Nauman Bin Ali, Jürgen Börstler, and Michael Unterkalmsteiner. 2019. Test-Case Quality–Understanding Practitioners' Perspectives. In *International Conference on Product-Focused Software Process Improvement*. Springer, 37–52.

[28] Christoph Treude and Martin P Robillard. 2016. Augmenting api documentation with insights from stack overflow. In *Proceedings of the International Conference on Software Engineering*. IEEE, 392–403.

[29] Michele Tufano, Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, Andrea De Lucia, and Denys Poshyvanyk. 2016. An empirical investigation into the nature of test smells. In *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 4–15.

[30] Arash Vahabzadeh, Amin Milani Fard, and Ali Mesbah. 2015. An Empirical Study of Bugs in Test Code. In *Proceedings of the 2015 IEEE International Conference on Software Maintenance and Evolution (ICSME) (ICSME '15)*. IEEE Computer Society, USA, 101–110. https://doi.org/10.1109/ICSM.2015.7332456

[31] Arie Van Deursen, Leon Moonen, Alex Van Den Bergh, and Gerard Kok. 2001. Refactoring test code. In *Proceedings of the 2nd international conference on extreme programming and flexible processes in software engineering (XP2001)*. 92–95.

[32] Matias Waterloo, Suzette Person, and Sebastian Elbaum. 2015. Test Analysis: Searching for Faults in Tests *(ASE '15)*. 149–154.

[33] Ian H Witten and Eibe Frank. 2002. Data mining: practical machine learning tools and techniques with Java implementations. *Acm Sigmod Record* 31, 1 (2002), 76–77.

[34] Sai Zhang, Darioush Jalali, Jochen Wuttke, Kivanç Muslu, Wing Lam, Michael D. Ernst, and David Notkin. 2014. Empirically revisiting the test independence assumption. In *International Symposium on Software Testing and Analysis, ISSTA '14, San Jose, CA, USA - July 21 - 26, 2014*. 385–396.