2021 IEEE/ACM 29th International Conference on Program Comprehension (ICPC) | 978-1-6654-1403-6/20/$31.00 ©2021 IEEE | DOI: 10.1109/ICPC52881.2021.00052

What is the Vocabulary of Flaky Tests? An Extended Replication

Bruno Henrique Pachulski Camara[1,2], Marco Aurélio Graciotto Silva[3], Andre T. Endo[4], Silvia Regina Vergilio[2]
[1]*Centro Universitário Integrado*, Campo Mourão, PR, Brazil
[2]*Department of Computer Science*, *Federal University of Paraná*, Curitiba, PR, Brazil
Email: bhpachulski@ufpr.br, silvia@inf.ufpr.br
[3]*Department of Computing*, *Federal University of Technology - Paraná*, Campo Mourão, PR, Brazil
Email: magsilva@utfpr.edu.br
[4]*Department of Computing*, *Federal University of Technology - Paraná*, Cornélio Procópio, PR, Brazil
Email: andreendo@utfpr.edu.br

***Abstract*—Software systems have been continuously evolved and delivered with high quality due to the widespread adoption of automated tests. A recurring issue hurting this scenario is the presence of flaky tests, a test case that may pass or fail non-deterministically. A promising, but yet lacking more empirical evidence, approach is to collect static data of automated tests and use them to predict their flakiness. In this paper, we conducted an empirical study to assess the use of code identifiers to predict test flakiness. To do so, we first replicate most parts of the previous study of Pinto et al. (MSR 2020). This replication was extended by using a different ML Python platform (Scikit-learn) and adding different learning algorithms in the analyses. Then, we validated the performance of trained models using datasets with other flaky tests and from different projects. We successfully replicated the results of Pinto et al. (2020), with minor differences using Scikit-learn; different algorithms had performance similar to the ones used previously. Concerning the validation, we noticed that the recall of the trained models was smaller, and classifiers presented a varying range of decreases. This was observed in both intra-project and inter-projects test flakiness prediction.**

***Index Terms*—test flakiness, regression testing, replication studies, machine learning**

I. Introduction

In regression testing, automated tests are run to validate whether changes and/or bug fixes do not have a negative impact on the software. Nevertheless, not all test failures in regression imply on faults in the production code [1]. Some tests have an intermittent behavior: they may pass or fail when executed in the same software version. Those tests cannot be trusted and are known as flaky tests [2].

Unfortunately, flaky tests are common. Eck et al. [3] surveyed 121 practitioners from Mozilla Foundation and reported test flakiness as a moderate to critical issue. Moreover, they found 58% of professionals had faced this problem in the last month. In such cases, 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 was a genuine regression failure and not an intermittent one [4].

This problem has brought attention form practitioners [5, 6, 7, 8] and researchers [9]. We can find some approaches to detect flaky tests. Most of them require that a lot of test cases are executed many times, [10, 11]. Because of this, static approaches have been proposed. They usually employ Machine Learning (ML) techniques [1, 4, 12] and are less costly. Among such approaches, we would like to highlight the one proposed by Pinto et al. [4] published in the Mining Software Repositories Conference (MSR2020). The approach automatically identifies flakiness based on a more comprehensive set of predictors than the others, and this is the reason why this study was chosen by us to be replicated. In addition to common characteristics of the test cases such as number of lines of code and occurrence of certain Java keywords, the used set of predictors is built based on the conjecture that there are some patterns in the test case code that could be used to automatically recognize flaky tests. Then a set of tokens are extracted and post-processed by applying some Natural Language Processing (NLP) techniques to compile a vocabulary of flaky tests.

Moreover, the work of Pinto et al. constructed and made available 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). To this dataset, five ML algorithms implemented by the Weka framework [13] were applied to predict flakiness with an F-measure of 0.95. The best prediction performance was obtained when using Random Forest and Support Vector Machine (SVM). In terms of the code identifiers that are most strongly associated with test flakiness, the authors found that words *job*, *action*, and *services* are commonly associated with flaky tests.

The results obtained by the original study are very promising, but we identified some threats regarding algorithms used and the generalization of the results to other projects. Then some questions not answered by the original work can be posed: 1) "Can we obtain similar results by using other different algorithms or the same algorithms with distinct implementations?", and 2) "Are the results valid for other datasets including different projects?". The goal of our replication

978-1-6654-1403-6/21/$31.00 ©2021 IEEE
DOI 10.1109/ICPC52881.2021.00052

study is to address such threats and questions.

Replication Studies have been gaining importance in software engineering. Works and reviews on this subject show an increasing number of replications published [14, 15, 16, 17]. Many conferences have included tracks dedicated to this subject. The community has highlighted the importance of producing adequate documentation to allow replication. Shull et al. [18] identified the types of replications. When researchers apply the same procedures to answer the same research questions as closely as possible the replication is called exact. When researchers investigate the same research question by using a different experimental procedure, it is called conceptual. Internal replications are conducted by the original researchers and external ones by an independent group [17]. They are dependent replications when researchers attempt to keep all the conditions of the experiment the same or very similar. Otherwise they are independent replications, when researchers deliberately vary one or more major aspects of the conditions of the experiment.

In this sense, our replication is external and exact. To answer our questions we applied the same procedures of original study, with some variations in the experimental conditions: research questions, algorithms, and datasets. First, we conducted a dependent replication, by adopting the same datasets and algorithms from framework Weka. This replication ensured a correct understanding of the original setup and confirmed original results. After, we performed an independent replication composed of two parts. In the first part, we used the same dataset but varying the algorithms implementation by adopting the framework Scikit-learn [19] and three additional algorithms. In the second part, we extended original work with additional research questions and performed cross-project validations to assess the performance of the trained models with different datasets, for intra- and inter-projects test flakiness prediction. This allows evaluating the generalization of the original results in real scenarios.

The main contributions of this paper are the following:

- Evaluation of three new algorithms: the results obtained by our replication using other ML framework such as Scikit-learn confirms through some evidence that the approach of original work, based on static detection of flaky tests, is effective. One of the new algorithms added, Logistic Regression, obtained the best value of recall, and best performance in the cross-project validation.
- Campbell and Stanley [20] argue that experiments need to be replicated in different contexts, at different times, and under different conditions before they can produce generalizable knowledge. Thus, replications like ours help investigating if the vocabulary of flaky tests obtained by the original work remains valid and can directly be transferred for other contexts like intra- and inter project test flakiness prediction.
- Implications of our findings to help researchers and developers in the challenging task of identifying flaky tests. Such implications may raise future research directions.
- A new repository with the procedures, datasets, and scripts generated from this replication, made available at https://github.com/bhpachulski/ICPC-RENE-Paper.

The paper is organized as follows. Section II reviews related work on flaky tests. Section III describes details of the original study following replication guidelines [21]. Section IV contains the setup of our replication study. Section V presents and analyses the obtained results. Section VI presents the main threats of our study. Section VII discusses some implications of our results. Section VIII presents our final remarks and concludes the paper.

II. Related Work

Test flakiness has a negative impact on the software development process: on one hand, debugging a yet-unknown flaky test may demand a huge effort as it is not actually a bug in the software; on the other hand, prematurely labeling (incorrectly) a test as flaky would let a bug escape to production and may harm the end-users.

Flaky tests are not only caused by changes in the software [2, 3]. For instance, Lam et al. [11] observed that more than half of analyzed flaky tests are order-dependent (OD), though other causes have been investigated [2]: asynchronous executions (Async Wait), concurrent execution (Concurrency), leakage resource (Resource Leak), communication (Network), tests based on date and time (Time), reading and writing files (IO), use of random data (Randomness), operations with floating-point numbers (Floating Point Operations) and use of unordered collections (Unordered Collections).

This problem has brought attention from practitioner and researchers. We noticed efforts on this subject in the industry [5, 6, 7, 8]. In the literature, we find works dedicated to this subject as reported by a recent review [9]. Dynamic approaches to detect flaky tests require re-execution of test cases usually fixing the number of times they will be executed [11, 22]. This is expensive and error-prone; the determination of this number is challenging, and an inadequate choice can lead to false negatives. As alternative, static approaches have been proposed [2, 3, 4, 12, 22, 23]. Most of these approaches are based on Machine Learning [4, 12, 23]. These last ones, more related to our replication study, are described below.

Memon et al. [23] propose to reduce the workload of the Google *Test Automation Platform* (TAP), avoiding the execution of tests with low probability of failure. Another goal is to present to developers insights about the project they are developing. As such, developers may take preemptive measures to prevent breakages. The proposed approach is based on a feature vector composed by Continuous Integration tool (CI), transitions from PASSED-to-FAILED, fixes (FAILED-to-PASSED), and programmers' activity in the version control tool. In this study, 2.07% of tests passed or failed at least once: 1.23% of them revealed faults introduced by developers, and 0.84%, namely 46,694 tests were flaky. The approach made it possible to reduce the number of executed test cases, without neglecting the fault detection capabilities.

King et al. [12] introduce an ML-based approach based on Bayesian networks to predict flaky tests. The problem is modeled like a disease in which symptoms can identify a flaky test. The authors propose a map of symptoms and causes, using static and dynamic metrics like Complexity Metrics (Test Assertion Count, Test Class/Method Size, Depth of Inheritance Tree), Implementation Coupling Metrics (Coupling Between Objects and Selector Stability Index), Non-Determinism Metrics (Cyclomatic Complexity and Explicit Wait Count), Performance Metrics (Average Execution Time), and General Stability Metrics (Failure Rate and Flip Rate). The evaluation was conducted with UI tests of a proprietary Web system; the tests are executed in a CI environment and five teams took part in the study. After three months, the supporting tool helped to reduce the test flakiness, in some cases up to 60%. Overall, the accuracy of approach's prediction was 65.7%. The features (symptoms) with the best prediction capabilities were High Test Complexity (88%) and Test Size (82%).

The approaches mentioned above present some advantages. The model derived is capable to predict flakiness with less cost, without re-executions of test cases. But among these approaches, the one from Pinto et al. [4] includes a more comprehensive set of predictors and presented promising results. Such an approach is the focus of our replication study and is detailed in the next section.

III. Original Study

The main hypothesis of the original study was that there are some syntactical patterns in the code of flaky test that can be used by NLP techniques to predict flakiness without code executions. Such patterns constitute the vocabulary of flaky tests. Below, we describe the main items regarding this study, following some guidelines for replication studies proposed by Carver et al. [21].

A. Approach

The adopted approach can be performed statically and includes the following steps:

1) Extraction of tokens: identifiers are extracted from test code labeled as flaky or non-flaky;
2) Processing of tokens: NLP techniques are applied, such as identifier splitting, stemming (removal of the suffix from a word), and stop word removal, to turn the extracted identifiers into tokens to be used as input for text classification algorithms. The identifiers are split using their camel-case syntax, and all resulting tokens are converted to lower case. Figure 1 contains an example of test case extracted from the original study [4] and its corresponding set of extracted tokens. The information about the tokens is then augmented with three numerical features, acting as proxies of code complexity: LOC: number of lines of code of the test case; keywords: the number of occurrences of each one of the 56 Java keywords the test case contains; and keyword count: total number of Java keywords present in the test case;
3) Flakiness prediction: five classifiers are applied on the resulting dataset using Weka [13]: Random Forest, Decision Tree (DT), Naive Bayes, Support Vector Machine (SVM), and Nearest Neighbour.

```
@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
```

Fig. 1. An example of test case and its tokenized result (extracted from Pinto et al. [4].)

B. Research questions

The original study investigated four research questions (RQs).

- **RQ1.** How prevalent and elusive are flaky tests? The goal of this question was twofold: 1) to confirm that flaky tests are very common in regression test suites, and 2) to analyse if there is an ideal number of reruns to be used to find flakiness.
- **RQ2.** How accurately can we predict test flakiness based on source code identifiers in the test cases? The goal of this question was to evaluate the performance of classifiers to predict test based on the source code identifiers without re-execution of the test suites.
- **RQ3.** What value do different features add to the classifier? The goal of this question was to evaluate the impact of the processing step (i.e. stemming and stop word removal) in the classifier performance, as well as some choices made by the authors, such as converting identifiers to numeric features and splitting identifiers.
- **RQ4.** Which test code identifiers are most strongly associated with test flakiness? The goal was to identify the test code identifiers which are more related to flakiness in order to help development, code review, and debugging tasks.

C. Dataset

The dataset was built based on 24 DeFlaker projects [10]. As this set of projects contains only information about flaky

test cases, the authors decided to execute the test suites of each project 100 times. The test case was flagged as non-flaky if a consistent outcome was obtained across all executions. In the end, an imbalanced set was obtained with a greater number of non-flaky tests. To deal with this problem, a number of non-flaky tests was selected, equal to the number of flaky tests in the original DeFlaker data set. This selection was performed in a way that the median sizes (in number of lines of code) of flaky and non-flaky tests were nearly the same. The data produced as result is available at: https://github.com/damorimRG/msr4flakiness/. This lab package, since now called msr4flakiness, is one of the datasets used in our replication study.

D. Evaluated metrics

To evaluate the performance of the classifiers, the authors split the data set into 80% for training and 20% for testing. They used 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), 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 visualizes the trade-off between true positive rate and false positive rate.

E. Summary of the results

As a result of RQ1, the authors found a low number of flaky tests, but flakiness is relatively common in IO-related projects. They concluded that detecting flakiness with test reruns is challenging. The number of executions may impact the detection since around 70% of the test cases identified as flaky passed in more than 90% of the runs.

Answering RQ2, all classifiers achieved very good performance in distinguishing flaky test cases from non-flaky test cases. Random Forest achieved the best precision (0.99), the SVM 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. The results are presented in Section V for comparison.

RQ3 investigates the performance impact of the different features used in the classifiers. To answer this question, the authors used the best classifiers: Random Forest (best precision and F_1-Score) and Support Vector Machine (best recall). The features evaluated were: no stemming, no stop word removal, no lower casing, no identifier splitting, only split identifier, no LOC, no Java keywords, no identifiers. The evaluation showed the impact on the classifier performance of most pre-processing steps was negligible. The only large impact was observed for Random Forest when Java keywords were included as tokens, but not identifier names. In this case, the performance would drop from an F_1-Score of 0.95 to 0.79. For SVM, 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.

To answer RQ4, the paper provides a list of 20 features with the highest information gain along with their frequency in the projects. The vocabulary associated with flaky tests contains words such as `job` (present only in 2 projects (out of 24)), `id` (appears in 9 projects), `table`, and `action`, many of which are associated with executing tasks remotely and/or using an event queue. Interestingly, the authors did not find a single token in the top 20 that was more strongly associated with non-flakiness. In that regard, for the Java keyword `throws` the authors conjecture that proper exception handling can help avoiding test flakiness.

F. Threats and Limitations

One of the main threats mentioned by the authors is the generalization of the results. The obtained vocabulary is limited to the Java language. But in addition to this, even if the same language is considered we would like to highlight two other threats. The first one is that the results may be valid only for the studied test cases, which are particular to the selected projects and their domains. A second possible threat is the ML algorithms used. To investigate the impact of such threats is the objective of our replication study.

IV. Replication Study Setup

We set out our replication study to address the threats of the original study mentioned in the last section. We can divide our study in two parts, each one of them corresponding to one of the identified threats and to a main research question.

RQ_1. Can we obtain similar results by using other ML algorithms or the same algorithms with distinct implementations?

- **$RQ_{1.1}$** How accurately can we predict test flakiness based on source code identifiers in the test cases?
- **$RQ_{1.2}$** What value of different features add to the classifier?
- **$RQ_{1.3}$** Which test code identifiers are most strongly associated with test flakiness?

Rationale. Answering RQ_1 we intend to investigate the influence of using other algorithms and implementations in the results of the original study. We believe the previous results are promising, yet further evidence would be desirable. For this end, replication studies are an important source of evidence [20]. Then, apart from the first RQ of the original study that aims to obtain the dataset, we used the dataset msr4flakiness (available lab package) to replicate the study for the remaining questions. As such, $RQ_{1.1}$, $RQ_{1.2}$, and $RQ_{1.3}$ are same ones investigated previously, but now using the framework Scikit-learn and applying three additional algorithms, successfully employed for processing text.

RQ_2. Are the results valid for other datasets including different projects?

- **RQ**$_{2.1}$ Can a trained classifier be successfully applied within the same projects (i.e., intra-project)?
- **RQ**$_{2.2}$ Can a trained classifier be successfully applied to other projects (i.e., inter-projects)?

Rationale. RQ$_2$ investigates if the vocabulary of flaky tests obtained by original study remains valid and can be directly used in other contexts. Answering this question, we intend to obtain evidence that the results can be generalized by varying the datasets used. For this end, we use different cross-project validation, not only using the original msr4flakiness. This question is herein proposed to assess the behavior of trained models towards the adoption by practitioners. In particular, we analyze two scenarios: *(i)* (RQ$_{2.1}$) there is a group of projects and historical data is used to predict flaky tests within this group (i.e., intra-project), and *(ii)* (RQ$_{2.2}$) to predict test flakiness in projects outside the group (i.e., inter-projects).

To answer **RQ**$_1$, we used the lab package msr4flakiness to replicate the original study in two steps. First like the work of Pinto et al., we adopted the ML framework Weka [13] to extract features, training and validation of ML algorithms. This step tries to reproduce the same results, as well as to know the intrinsic configurations of feature extraction, training and tests. The idea is to confirm previous results and understand the available package msr4flakiness. As a result of this dependent replication, apart from RQ1, which is not directly related to the goal of our study, we obtained the same results of the original study for all questions. These results are not presented and discussed in the present paper but they are in our replication package.

In this way, as a result of this first step we ensure that we are interpreting well and adopting the same procedure of original study. Then in the second step and considering the goals of **RQ**$_1$, we use a different ML framework and algorithms, so we chose Scikit-learn [19]. Scikit-learn is one of the most popular ML libraries[1], providing a platform to build ML applications using the programming language Python. This makes it possible to evaluate if the results could be reproduced in a different software platform, and facilitates future extensions of the study, once Python seems to be the de facto programming language for ML [8].

Targeting RQ$_{1.1}$, we first loaded the dataset msr4flakiness and ran the pre-processing of features (tokens). For feature extraction, we adopted the method CountVectorizer of Scikit-learn along with a handcrafted tokenizer, shown in Figure 2, to replicate the same tokenization criteria used in Weka at the original study.

```
def weka_tokenizer(string):
    delimiters =
        re.compile("[ |\n|\f|\r|\t|.|,|;|:|'|\"|(|)|?|!]")
    return list(filter(None, delimiters.split(string)))
```

Fig. 2. Function for word separation in Python.

[1]https://github.blog/2019-01-24-the-state-of-the-octoverse-machine-learning/

Scikit-learn has implementations of the five classifiers used in the original study: Random Forest, Decision Tree (DT), Naive Bayes, Support Vector Machine (SVM) and Nearest Neighbour. Yet, we needed to adjust them to use the same parameters of Weka to obtain the same results. Furthermore, we included three different classifiers: Logistic Regression (LR), Linear Discriminant Analysis (LDA) and Perceptron. The choice of these algorithms is based on successful studies that use linear classifiers and neural networks for text classification [24, 25].

We used 80% of the dataset for training and 20% for validation, following the original study. Figure 3 shows how the classifiers were parameterized in Scikit-learn based on the original study Weka parameters. We used the LinearSVC implementation of SVM.

```
'random_forest': RandomForestClassifier(random_state=1),
'decision_tree': DecisionTreeClassifier(min_samples_leaf=1)
    ,
'naiveb_bayes': GaussianNB(),
'svm': CalibratedClassifierCV(LinearSVC(fit_intercept=False
    , tol=0.001, C=1, dual=False, max_iter=100000), method=
    'sigmoid'),
'knn': KNeighborsClassifier(n_neighbors=1, metric='
    euclidean'),
'LR': LogisticRegression(max_iter=1000),
'perceptron': CalibratedClassifierCV(Perceptron()),
'LDA': LinearDiscriminantAnalysis()
```

Fig. 3. Classifiers' parameters used in Scikit-learn.

Concerning RQ$_{1.2}$, the goal is to evaluate the impact of pre-processing steps for word separation and feature transformation. The following configurations were adopted considering datasets made available by the msr4flakiness experimental package: No Stemming, No Stop Words Removal, No Lowercasing, No Identifier Split, Only Split Identifiers, No Lines of Code, No Java Keywords, No Identifiers. These configurations were evaluated with the same algorithms of the original study to allow comparison.

In RQ$_{1.3}$, we evaluated the impact of features using the method `mutual_info_classif` of Scikit-learn with default settings, which is equivalent to the entropy calculation of Weka. At this step we considered the entire dataset. The information gain (as known as entropy) is calculated for each output variable. This value ranges from 0 (no gain) to 1 (maximum of information gain). All these aforementioned steps were performed as described in the previous work.

Concerning **RQ**$_2$, a different dataset is required to validate the trained models. Therefore, we selected the 335 flaky tests from 72 different projects, collected in Lam et al. [11]; we refer to this dataset as iDFlakies. This dataset does not contain examples of non-flaky tests, so only recall will be used to analyze the results. To obtain the dataset's features, we used the same process performed by the original study based on the scripts for data extraction. To support this step, we adopted the eight classifiers trained for answering RQ$_1$ using Scikit-learn.

RQ$_{2.1}$ focuses on the intra-project scenario. For this question, we generated from iDFlakies a validation dataset using the process provided by the authors of the original study. As

a result we transformed the flaky test cases into a dataset that comprises all the features used in the original study. Then we filtered out duplicate tests and tests from projects not present in dataset msr4flakiness. So, we are testing the scenario in which the historical data of a set of projects is used to predict test flakiness within this same set of projects. In the end, the validation set for this question contains only flaky tests (80) from 22 different projects.

In $RQ_{2.2}$, we evaluated the inter-project scenario. So, we generated from iDFlakies a validation dataset by filtering out tests from projects present in the dataset msr4flakiness. So, we are testing the scenario in which the historical data of a set of projects is used to predict test flakiness in a different set of projects. At the end, the validation set for this question contains only flaky tests (256) from 64 different projects.

V. Analysis of Results

This section analyses the obtained results in order to answer the posed RQs.

A. RQ_1 – Replicating the study with Scikit-learn

Below we compare the results of our replication using Scikit-learn with the ones obtained in the original study using Weka. In our analysis we adopted the same evaluation metrics used in the original study (see Section III): precision, recall, F_1-score, MCC and AUC.

1) $RQ_{1.1}$ – How accurately can we predict test flakiness based on source code identifiers in the test cases?: The results of our replication using Scikit-learn were similar to the results of the original study (which used Weka), as shown in Table I. The greatest difference was for the Nearest Neighbour classifier, with a -0.7% difference for Scikit-learn considering Recall, and Naive Bayes, with a 0.8% difference for Scikit-learn regarding MCC.

The Random Forest algorithm had the highest score for all evaluated metrics in our replication study considering the algorithms of the original study. It correctly identified 98% of all the flaky tests classified by the model. The results for MCC of 0.90 and for F_1-score of 0.94 show that the quality of the prediction model is very good. The high score for both MCC (0.90) and F_1-score (0.94) were expected, as the dataset was balanced. AUC for Random Forest was also the highest (0.98), providing further evidence regarding the model classification quality. Considering the extended set of classifiers, Random Forest had a lower score only for Recall when considering the Logistic Regression classifier using the Scikit-learn: 0.90 versus 0.91, respectively.

For the new classifiers considered in our study, Logistic regression provided comparable results to Random Forest, with better results for recall. Nonetheless, the difference was of just two percentage points. Perceptron had a similar performance to Logistic Regression. Regarding LDA, it performed similarly to Decision Tree, with a lower overall performance than the other algorithms.

Answer to $RQ_{1.1}$: The classifiers performed very well, similarly to the original study. The difference of the results due to machine learning frameworks is small. The additional classifiers obtained results similar to the ones investigated previously. LR presented the best recall value.

2) $RQ_{1.2}$ – What value of different features add to the classifier?: Although every classifier has performed very well, we considered models created with all the features extracted from the dataset. However, it is important to analyse the impact of each feature and pre-processing applied to them. Pinto et al. [4] considered three types of features: identifiers, Java keywords, and LOC metric. For the identifiers, which comprise most of the features considered in the model, several pre-processing steps were applied: stemming, stop word removal, lowercasing, splitting. In Table II, we present the performance of Random Forest and SVM classifiers with respect to precision, recall, F_1-score, MCC and AUC, with different subsets of features and pre-processing configurations.

As in the original study, the exclusion of tokens from the generated model (No Identifiers) had the greatest impact in the classifier performance in the replication. The precision reduces drastically, from 0.98 to 0.72 when using Random Forest and from 0.94 to 0.59 using SVM. Considering that identifiers are directly related to the vocabulary in each test case, this result was expected. Interestingly, disabling pre-processing for identifiers did not significantly changed the performance of the model. Similarly, excluding LOC or Java keywords other features also have minor impact, with a variation of less than three percentage points for SVM.

Answer to $RQ_{1.2}$: Similarly to the original study, different features did not show much impact in the classifiers, except when tokens (No Identifiers) extracted from test cases were not considered.

3) $RQ_{1.3}$ – Which test code identifiers are most strongly associated with test flakiness?: The association of features and test code identifiers was evaluated by calculating the information gain based on the entropy of the features extracted after pre-processing the text of the test cases. In Table III we report the features with higher information gain for original study and our replication. The value for the information gain of the studies are different. For instance, for `job` the original study reported 0.2053 while we got 0.1449 for the replication. Nonetheless, the top three features are the same for both models and, considering the top twenty features, the only differences are the feature `coordinatorjob` (present in the original study, but absent from the replication) and `get` (present in the replication, but absent in the original study).

Thus, despite the different values for information gain, we can conclude that features associated with test flakiness are the mostly same of the original study. The features are associated with executing and coordinating tasks (`job`, `services`, `action`, `coordinator`, `workflow`, `getstatus`), persistance (`table`, `id`, `record`, `jpa`, `jpaservice`). The most relevant difference was with respect to the feature `throws (keyword)`. Although it was identically ranked, in the original study it was related to few tests (10), in the replication it was exactly the opposite, with 2.348 test

cases. We also conjecture that exception handling is related to flakiness.

Answer to $RQ_{1.3}$: The flaky vocabulary is related to executing and coordinating tasks, and persistence. The set of words is very similar to the one of original study.

Summary of RQ_1: Our replication gives more evidence about the validity of the original results using other ML framework and algorithms. The results obtained are very similar, concerning the performance of the algorithms, value of the used features and vocabulary of flaky tests found. It is worth emphasizing the good performance of the additional classifiers. LR reached the best values of Recall and second best values of AUC. Other point to be highlighted is that the lab package provided by the original study is easy-to-use and self-contained, which allows us to replicate the study without contacting the authors.

B. RQ_2 – Cross-project validation

For RQ_2, we extended the original study by addressing the performance of the classification model using the vocabulary of flaky tests with different datasets. First, we consider using the trained classifier within a different set of test cases from the same software projects (intra-projects). Second, we applied the classifier to other projects (inter-projects).

1) $RQ_{2.1}$ – Can a trained classifier be successfully applied within the same projects (i.e., intra-project)?: The intra-project validation was performed with the dataset of iDFlakies considering only the 22 projects which are also in the dataset msr4flakiness. The results are presented in Table IV. As previously described, this dataset does not have tests classified as non-flaky, thus the confusion matrix has only true positives (TP) and false negatives (FN). Thus, we just considered Recall to evaluate the performance of the classifiers.

The classifiers performance was not satisfactory compared to the results for msr4flakiness in the $RQ_{1.1}$ (Table I). Random Forest was the classifier with the best performance regarding Recall on msr4flakiness (0.90), but achieved only 0.08 for the intra-project scenario. SVM also experienced a lower recall, dropping from 0.86 to 0.29. Interestingly, the classifiers we had added for the extended replication performed better than the original ones, although they also had

TABLE I
CLASSIFIER PERFORMANCE.

(a) Original study					
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
SVM	0.93	**0.92**	0.93	0.85	0.93
Nearest Neighbour	0.97	0.88	0.92	0.85	0.93

(b) Replication study					
Algorithm	**Precision**	**Recall**	**F_1**	**MCC**	**AUC**
Random Forest	**0.98**	0.89	**0.94**	**0.89**	**0.98**
Decision Tree	0.87	0.86	0.86	0.74	0.87
Naive Bayes	0.95	0.84	0.89	0.81	0.90
SVM	0.93	0.86	0.90	0.81	0.96
Nearest Neighbour	**0.98**	0.81	0.89	0.81	0.90
Perceptron	0.95	0.83	0.88	0.81	0.96
LR	0.91	**0.91**	0.91	0.84	0.96
LDA	0.83	0.78	0.80	0.63	0.87

TABLE II
PERFORMANCE OF RQ 1.1 CLASSIFIERS WITHOUT FEATURES.

Original study					
(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
(c) SVM					
Features	**Precision**	**Recall**	**F1**	**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	0.54	0.87	0.74	0.40	0.68

Replication study					
(b) Random Forest					
Features	**Precision**	**Recall**	**F1**	**MCC**	**AUC**
All Features	0.98	0.90	0.94	0.90	0.98
No Stemming	0.97	0.89	0.93	0.88	0.98
No Stop W. Removal	0.98	0.89	0.93	0.87	0.98
No Lowercasing	0.98	0.90	0.93	0.88	0.98
No Identifier Split.	0.98	0.90	0.94	0.89	0.98
Only Split Identif.	0.97	0.90	0.93	0.88	0.98
No Lines of Code	0.98	0.89	0.93	0.88	0.98
No Java Keywords	0.98	0.89	0.93	0.88	0.98
No Identifiers	0.72	0.81	0.76	0.52	0.85
(d) SVM					
Features	**Precision**	**Recall**	**F1**	**MCC**	**AUC**
All Features	0.94	0.86	0.90	0.82	0.96
No Stemming	0.94	0.85	0.89	0.81	0.96
No Stop W. Removal	0.93	0.86	0.89	0.81	0.96
No Lowercasing	0.93	0.86	0.90	0.81	0.96
No Identifier Split.	0.94	0.88	0.91	0.84	0.97
Only Split Identif.	0.92	0.90	0.91	0.83	0.96
No Lines of Code	0.94	0.86	0.90	0.80	0.96
No Java Keywords	0.92	0.89	0.91	0.82	0.96
No Identifiers	0.59	0.83	0.69	0.32	0.73

TABLE III
TOP 20 FEATURES BY INFORMATION GAIN.

(a) Original study				
Feature	**Inf. Gain**	**Tests**	**Flaky tests**	**Non-Flaky tests**
job	0.2053	528	524	4
table	0.1449	414	406	8
id	0.1419	584	522	62
action	0.1365	395	387	8
oozie	0.1359	274	274	0
services	0.1309	378	371	7
coord	0.1192	307	307	0
getid	0.1076	288	287	1
coordinator	0.1070	258	258	0
xml	0.1061	253	247	6
loc	0.0977	-	-	-
workflow	0.0913	207	207	0
getstatus	0.0884	248	246	2
throws_keyword	0.0873	10	3	7
record	0.0845	314	296	18
jpa	0.0780	207	207	0
jpaservice	0.0752	200	200	0
service	0.0733	434	367	67
wf	0.0721	192	192	0
coordinatorjob	0.0689	184	184	0

(b) Replication study				
Feature	**Inf. Gain**	**Tests**	**Flaky tests**	**Non-Flaky tests**
job	0.1449	530	525	5
table	0.1029	414	406	8
id	0.1004	577	525	52
services	0.0977	378	371	7
action	0.0972	396	388	8
oozie	0.0942	346	346	0
loc	0.0879	-	-	-
coord	0.0826	307	307	0
xml	0.0752	356	341	15
getid	0.0746	288	287	1
coordinator	0.0741	278	278	0
get	0.0691	2194	1260	934
workflow	0.0633	240	240	0
throws_keyword	0.0615	2348	1327	1021
getstatus	0.0613	248	246	2
record	0.0596	314	296	18
service	0.0590	451	383	68
jpa	0.0541	207	207	0
jpaservice	0.0521	200	200	0
wf	0.0499	192	192	0

a lower performance. Linear Discriminant Analysis (LDA) had a recall of 0.75, followed by Logistic Regression (LR), with 0.68.

TABLE IV
CLASSIFIERS PERFORMANCE IN THE INTRA-PROJECT SCENARIO.

Classifier	**Recall**	**TP**	**FN**
LDA	0.75	60	20
LR	0.68	54	26
Perceptron	0.51	41	39
SVM	0.29	23	57
Decision Tree	0.19	15	65
KNN	0.19	15	65
Random Forest	0.08	6	74
Naive Bayes	0.11	9	71

Regarding the importance of each feature of the classification, we calculated the information gain and reported the top-20 features in the Table V. The only feature common to msr4flakiness and the intra-project dataset was `loc`, but with no actual information. There some features related to execution and coordination of tasks (`init`, `createdirwithhttp`), but most are related to I/O operations (`reader`, `write`, `directory`, `folder`). The feature `public`, a Java keyword, had the highest information gain (0.8188) and is associated to 80 flaky tests. As most (if not all) tests are declared in public methods, that is not actually unexpected, although it should not be relevant to detect flaky tests. Another interesting fact is that, although we reported top-20 features in Table V, only 15 had an information gain greater than zero: the five features with no information gain were LOC and Java keywords (`abstract`, `assert`, `boolean` and `break`).

TABLE V
TOP 20 FEATURES BY INFORMATION GAIN IN THE INTRA-PROJECT SCENARIO.

Feature	**Inf. Gain**	**Tests Flaky**
public_keyword	0.8188	80
acl	0.8188	6
created	0.8188	6
reader	0.8188	6
createdirwithhttp	0.0791	6
createsnapshot	0.0791	6
directory	0.0791	6
folder	0.0791	6
gethadoopusers	0.0791	6
hadoopusersconftesthelper	0.0791	6
init	0.0791	6
no	0.0791	6
touri	0.0791	6
write	0.0791	6
assertnotnull	1.1102	28
loc	0.0	-
abstract_keyword	0.0	0
assert_keyword	0.0	68
boolean_keyword	0.0	0
break_keyword	0.0	0

Answer to $RQ_{2.1}$: The performance of the classification model to identify flaky tests within the same projects was low. The features with higher information gain are distinct from those previously identified, and they are related to I/O operations, execution and coordination of tasks, and Java keywords.

2) $RQ_{2.2}$ – Can a trained classifier be successfully applied to other projects (i.e., inter-projects)?: To answer $RQ_{2.2}$, we used the classification models trained for $RQ_{1.1}$ to test the inter-project dataset. This dataset has 256 flaky tests of 64 projects distinct from those from msr4flakiness. Similarly to $RQ_{2.1}$, we considered only recall to evaluate the classifier performance as the inter-project dataset contains just flaky tests.

The performance of the classifiers for the inter-projects dataset was very low, as detailed in table VI. Considering the Decision Tree classifier, which achieved a recall of 0.86 in $RQ_{1.1}$, the result for the inter-project dataset was of 0.16. Yet, from the five models considered in the original study, it was the best classifier. Random Forest, which had a recall of 0.90 for in $RQ_{1.1}$, got only 0.02, correctly identifying just four flaky tests. In the context of our extended replication study, LDA was the best, with a recall of 0.48.

It is worth noticing that the performance for inter-projects was significantly lower than for the intra-project scenario. For instance, LDA had a recall of 0.76 in the latter against 0.48 in the former.

TABLE VI
CLASSIFIERS PERFORMANCE IN THE INTER-PROJECT SCENARIO.

Classifier	Recall	TP	FN
LDA	0.48	122	134
LR	0.40	102	154
Perceptron	0.38	97	159
Decision Tree	0.16	40	216
SVM	0.11	27	229
Naive Bayes	0.09	24	232
KNN	0.09	22	234
Random Forest	0.02	4	252

The information gain for the features extracted from the inter-project dataset is shown in Table VII. The features are different from those of $RQ_{1.1}$ and intra-project dataset in $RQ_{2.1}$. The features that are more often associated to flaky tests are related to asynchronous calls (`await`, `export`, `handler`, `protocol`, `server`, `task`, `taskpayloadbuilder`, `url`) or they are Java keywords (`return`, `try`). Strikingly, the special character { is among the top-20 features, even though it is present in every flaky test and could not provide any relevant information to classify a test case as flaky.

Answer to $RQ_{2.2}$: Classifiers trained with the dataset msr4flakiness could not provide a satisfactory performance when testing a dataset of different projects (inter-projects). The features with higher information gain are related to asynchronous calls and they are distinct from those previously identified.

Summary of RQ_2: We obtained some evidence that the vocabulary of flaky tests from original study cannot be directly used in other contexts. In general, training the classifiers with the dataset msr4flakiness does not lead a good prediction in both scenarios evaluated: intra- and inter-projects. A possible reason for this is overfitting, caused by excessive number of tokens in the vocabulary and few examples.

TABLE VII
TOP 20 FEATURES BY INFORMATION GAIN IN THE INTER-PROJECT SCENARIO.

Feature	Inf. Gain	Tests Flaky
getname	0.8882	22
namingcontext	0.8882	22
same	0.8882	22
await	0.8674	14
export	0.8674	14
return_keyword	0.8465	17
be	0.8465	17
handler	0.8465	30
is	0.8465	17
this_keyword	0.8396	14
protocol	0.8396	14
}	0.8396	14
collections	0.8188	17
context	0.8188	19
server	0.8119	14
task	0.8049	19
of	0.7633	19
taskpayloadbuilder	0.7633	19
url	0.6939	40
try_keyword	0.6661	51

VI. THREATS TO VALIDITY

Threats to construct validity are related to metrics used to evaluate the results. As a replication, the same metrics of Pinto et al. [4] were employed to support the comparisons. As such, this study is subjected to the same threats that are the evaluation method based on precision and F_1-Score. For RQ_2, a dataset only with flaky tests was used. So, the precision was always the maximum and we did not consider it in our analyses. This limitation should be addressed in future work.

Threats to internal validity may comprise the results when relating independent and dependent variables. To mitigate this, we carefully replicate the experiments so that the same results were achieved with the Weka software. With Scikit-learn, we tried our best to replicate the results, while slightly differences were noticed due to mostly the feature extraction.

External validity is connected to the generalization of obtained results. While the replication brings more evidence and the adoption of different datasets helps to evaluate different scenarios, as in the original study we cannot generalize the results, we are limited to a Java language and limited domain projects. When looking at the intra-project and inter-project contexts, the results herein presented had a meaningful reduction of performance, the fact that motivates further research. The sample used to evaluate the classifiers is small and should be increased to better understand the possibilities.

VII. DISCUSSION

This section discusses some implications of our findings, which may guide new research in the area.

The use of tokens extracted from test cases was effective in identifying flaky tests for the first part of our study. However, for the extended replication, considering test cases for the same projects (which should share the vocabulary) and for different projects (which may not hold such assumption), the recall was rather low. This suggests an overfitting of the classification model.

The choice of features or models that are more generalizable and useful in broader contexts could be an alternative to improve the performance of classification models for flaky tests on intra and inter-projects scenarios. Several approaches could be analyzed to address this issue. For instance, we could refine a more general vocabulary, removing project-specific words. As reported, stop words had a minor impact on the classifier's performance. However, that could be due to an ineffective list of words. As we can see in Table VII, there are features which are usually considered as stop words, such as common auxiliary verbs (`be`, `is`), prepositions (`of`), and symbols (}). We could also employ term weighting (for example, TF-IDF) to reduce the impact of features common to a single project or to the entire dataset.

After inspecting the extracted features and the related test cases, we observed that the current tokenizer is splitting dates, URLs, and filenames. For usual text mining and NLP-based approaches, such heuristic would have minor impact on the classifier performance, but that information is relevant for flaky tests, which can be associated to I/O and time/calendar operations [2, 26]. An improved parser and tokenizer could address such cases. Some tokens, like symbol "{" in Table VII, being recognized as a feature point to issues regarding feature engineering. It is important to emphasize that such special cases are also present in the original study.

Currently, we extracted features only from the body of the test cases, but tests can become flaky due to changes unrelated to that information [27]. Thus we could consider external information associated to each test case, such as helper methods and library dependencies. Order-dependent flaky tests are often detected in test suites with helper methods [22] that configure the state of the application. For Java applications these fixture methods are annotated with `BeforeEach`, `BeforeClass`, `AfterEach`, and `AfterClass`, defining the test workflow. The association of such annotations to order-dependent flaky tests makes annotations a candidate for feature selection. Approximately 50% of the flaky tests for the iDFlakies dataset considered for $RQ_{2.2}$ are order-dependent [11]. The absence of features associated to test workflow can partially explain the poor performance of the classifier for that dataset.

Version control systems may also be an interesting source of data, such as code churn, modified code, commits, and contributors. Finally, dynamic features like code coverage, monitoring data, and test execution reports deserve further investigation.

VIII. Conclusions

Flaky tests are intermittently passing or failing, causing distrust in test automation. This may harm the software development process of several companies that rely on automated tests to support a continuous integration and delivery environment. Therefore, the prevention and identification of potential flaky tests are investigated by practitioners and researchers.

This paper intends to increase the body of evidence on using code identifiers to predict flaky tests. To do so, we conducted an extended replication of Pinto et al. [4]' study on the vocabulary of flaky tests. Besides replicating the previous results, we extended it by using a different ML framework (namely, Scikit-learn), assessing different classifiers, and validating the trained models with different datasets for intra- and inter-project contexts.

The results obtained in this work demonstrated that the proposal to create a vocabulary of flaky tests by the authors of the original study has a process of extracting characteristics, training is replicable and extensible, thus reducing possible bias added by researchers, process and software. The results obtained during the intra- and inter-project tests demonstrated that the defined vocabulary does not have the level of generalization sufficient to predict flaky tests with the same performance obtained during the tests of the classifiers. We observed that the context of a given project has a major impact on the vocabulary of flaky tests; this may hinder the adoption of code identifiers to predict test flakiness.

A potential future work would be to investigate a subset of existing vocabularies that generalizes for different projects. Other direction is to evaluate if the prediction of test flakiness may benefit from different features used in other studies [12, 23], along with code identifiers.

Acknowledgment

This work is partially supported by CNPq (Andre T. Endo grant nr. 420363/2018-1 and Silvia Regina Vergilio grant nr. 305968/2018-1) and by IN2 Institute (Bruno Henrique Pachulski Camara grant nr. 002/2021).

References

[1] K. Herzig and N. Nagappan, "Empirically detecting false test alarms using association rules," in *37th International Conference on Software Engineering*. Piscataway, NJ, EUA: IEEE, May 2015, pp. 39–48.

[2] Q. Luo, F. Hariri, L. Eloussi, and D. Marinov, "An empirical analysis of flaky tests," in *22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. New York, NY, EUA: ACM, Nov. 2014, pp. 643–653.

[3] M. Eck, F. Palomba, M. Castelluccio, and A. Bacchelli, "Understanding flaky tests: the developer's perspective," in *27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. New York, NY, EUA: ACM, Aug. 2019, pp. 830–840.

[4] G. Pinto, B. Miranda, S. Dissanayake, M. d'Amorim, C. Treude, and A. Bertolino, "What is the vocabulary of flaky tests?" in *17th International Conference on Mining*

Software Repositories. Seoul, South Korea: IEEE / ACM, Jun. 2020.

[5] J. Micco, "Flaky tests at Google and how we mitigate them," Web page, May 2016. [Online]. Available: https://bit.ly/2Nz4fF5

[6] M. Fowler. (2011, Apr.) Eradicating non-determinism in tests. Web page. Accessed: 2021-02-16. [Online]. Available: https://bit.ly/2ZlJ63W

[7] J. Palmer. (2019, May) Test Flakiness – methods for identifying and dealing with flaky tests. Web page. Accessed: 2021-02-16. [Online]. Available: https://bit.ly/2NbesYv

[8] W. Goddard. (2018, Oct.) What's the best programming language for machine learning applications? Web page. Accessed: 2021-02-16. [Online]. Available: https://cutt.ly/Hk1flto

[9] Zolfaghari, Behrouz, P. R. M., G. Srivastava, and Y. Hailemariam, "Root causing, detecting, and fixing flaky tests: State of the art and future roadmap," *Software: Practice and Experience*, 2020. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.2929

[10] J. Bell, O. Legunsen, M. Hilton, L. Eloussi, T. Yung, and D. Marinov, "DeFlaker: Automatically detecting flaky tests," in *40th International Conference on Software Engineering*. New York, NY, EUA: ACM, May–Jun. 2018, pp. 433–444.

[11] W. Lam, R. Oei, A. Shi, D. Marinov, and T. Xie, "iDFlakies: A framework for detecting and partially classifying flaky tests," in *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*. IEEE, Apr. 2019, pp. 312–322.

[12] T. M. King, D. Santiago, J. Phillips, and P. J. Clarke, "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, Jul. 2018, pp. 100–107.

[13] I. H. Witten, E. Frank, L. Trigg, M. Hall, G. Holmes, and S. J. Cunningham, "Weka: Practical machine learning tools and techniques with java implementations," 1999.

[14] R. M. M. Bezerra, F. Q. B. da Silva, A. M. Santana, C. V. C. Magalhaes, and R. E. S. Santos, "Replication of empirical studies in software engineering: An update of a systematic mapping study," in *2015 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, 2015, pp. 1–4.

[15] B. Kitchenham, "The role of replications in empirical software engineering—a word of warning," *Empirical Software Engineering*, vol. 13, pp. 219–221, 04 2008.

[16] M. Shepperd, N. Ajienka, and S. Counsell, "The role and value of replication in empirical software engineering results," *Information and Software Technology*, vol. 99, pp. 120–132, 2018. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0950584917304305

[17] F. Silva, M. Suassuna, C. França, A. Grubb, T. Gouveia, C. Monteiro, and I. Santos, "Replication of empirical studies in software engineering research: A systematic mapping study," *Empirical Software Engineering*, vol. 19, 09 2012.

[18] F. J. Shull, J. C. Carver, S. Vegas, and N. Juristo, "The role of replications in empirical software engineering," *Empirical Software Engineering*, vol. 13, pp. 211–218, Apr. 2008.

[19] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and Édouard Duchesnay, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, no. 85, pp. 2825–2830, 2011.

[20] D. T. Campbell and J. C. Stanley, *Experimental and quasi-experimental designs for research*, 1st ed. Chicago, IL, EUA: Houghton Mifflin, 1963.

[21] J. C. Carver, "Towards reporting guidelines for experimental replications: A proposal," in *1st International Workshop on Replication in Empirical Software Engineering Research (RESER)*, 2010.

[22] A. Shi, W. Lam, R. Oei, T. Xie, and D. Marinov, "iFixFlakies: A framework for automatically fixing order-dependent flaky tests," in *27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. New York, NY, EUA: ACM, Aug. 2019, pp. 545–555.

[23] A. M. Memon, Z. Gao, B. N. Nguyen, S. Dhanda, E. Nickell, R. Siemborski, and J. Micco, "Taming google-scale continuous testing," in *39th IEEE/ACM International Conference on Software Engineering: Software Engineering in Practice Track, ICSE-SEIP 2017, Buenos Aires, Argentina, May 20-28, 2017*. IEEE Computer Society, 2017, pp. 233–242.

[24] S. Dreiseitl and L. Ohno-Machado, "Logistic regression and artificial neural network classification models: a methodology review," *Journal of Biomedical Informatics*, vol. 35, no. 5, pp. 352–359, 2002.

[25] T. Pranckevičius and V. Marcinkevičius, "Comparison of naive bayes, random forest, decision tree, support vector machines, and logistic regression classifiers for text reviews classification," *Baltic Journal of Modern Computing*, vol. 5, no. 2, p. 221, 2017.

[26] W. Lam, P. Godefroid, S. Nath, A. Santhiar, and S. Thummalapenta, "Root causing flaky tests in a large-scale industrial setting," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*. New York, NY, EUA: ACM, Jul. 2019, pp. 101–111.

[27] W. Lam, K. Muslu, H. Sajnani, and S. Thummalapenta, "A study on the lifecycle of flaky tests," in *ICSE '20: 42nd International Conference on Software Engineering, Seoul, South Korea, 27 June - 19 July, 2020*, G. Rothermel and D. Bae, Eds. ACM, 2020, pp. 1471–1482.