A Replication Study on the Usability of Code Vocabulary in Predicting Flaky Tests

Guillaume Haben
University of Luxembourg
guillaume.haben@uni.lu

Sarra Habchi
University of Luxembourg
sarra.habchi@uni.lu

Mike Papadakis
University of Luxembourg
michail.papadakis@uni.lu

Maxime Cordy
University of Luxembourg
maxime.cordy@uni.lu

Yves Le Traon
University of Luxembourg
yves.letraon@uni.lu

***Abstract*—Industrial reports indicate that flaky tests are one of the primary concerns of software testing mainly due to the false signals they provide. To deal with this issue, researchers have developed tools and techniques aiming at (automatically) identifying flaky tests with encouraging results. However, to reach industrial adoption and practice, these techniques need to be replicated and evaluated extensively on multiple datasets, occasions and settings. In view of this, we perform a replication study of a recently proposed method that predicts flaky tests based on their vocabulary. We thus replicate the original study on three different dimensions. First, we replicate the approach on the same subjects as in the original study but using a different evaluation methodology, *i.e.*, we adopt a time-sensitive selection of training and test sets to better reflect the envisioned use case. Second, we consolidate the findings of the initial study by building a new dataset of 837 flaky tests from 9 projects in a different programming language, *i.e.*, Python while the original study was in Java, which comforts the generalisability of the results. Third, we propose an extension to the original approach by experimenting with different features extracted from the Code Under Test. We find that a more robust validation consistently decreases performance on the reported results of the original study, but, fortunately, the model remains capable to decently predict flaky tests. We find re-assuring results that the vocabulary-based models can also be used to predict test flakiness in Python. Finally, we find that the information lying in the Code Under Test has a limited impact on the performance of the vocabulary-based models.**

***Index Terms*—Software testing, regression testing, flakiness**

I. Introduction

Regression testing is an important step of software development that ensures the stability of existing software features and allow multiple developers to work on a shared codebase. In typical large scale development workflows, test suites are run after code changes to highlight any misbehaviour and validate new software releases. Unfortunately, software tests do not always give consistent results. This inconsistent behaviour is often referred to as test flakiness. Flaky tests exhibit a non-deterministic nature, *i.e.* they pass and fail for the same version of a program and the test [1].

Flakiness plagues Continuous Integration (CI) as tests are generally expected to pass in order to merge code changes [2]. Thus, flakiness introduces uncertainty, meaning that testers cannot be sure whether failures are true or not. Besides, flakiness affects productivity as developers invest time in reproducing and debugging flaky failures. Developers may also lose trust in their test suite and stop relying on it if there are too many false signals. Consequently, they could ignore failing tests that are caused by real defects in the program.

Several studies and reports from industrial actors have highlighted the prevalence and impact of flakiness [3]–[5]. For instance, at Google, there are 150 million test executions per day, and almost 16% of their 4.2 million test cases have some level of flakiness [6]. Perhaps worse, over 80% of observed transitions (false Failures or Passes) at Google workflow are caused by flaky tests [4], indicating an important level of uncertainty in the test signal.

A common approach to deal with flakiness is to re-run a failing test several times, hoping to expose non-deterministic behaviour. Unfortunately, these reruns imply cost both computationally- and time-wise. In the case of Google, this leads the company to spend between 2 and 16% of its computer resources rerunning flaky tests [6]. Many other companies report having problems dealing with flaky tests, including Huawei [7], Mozilla [8], Facebook [9] and Spotify [10].

To mitigate this problem, several strategies have been developed to detect flakiness. These can be divided into two main categories; the *dynamic approaches* that involve running the tests and analysing their outputs and logs over time, and the *static approaches* that attempt to identify flaky tests without any test execution.

Micco and Memon [11] presented a dynamic approach that identifies flaky tests by looking for specific patterns at the test execution outcomes observed in the recent development history (Pass to Fail, Fail to Pass), thereby proposing a simple pattern matching approach that achieves a 90% accuracy in classifying tests [11]. A similar approach was also presented by Apple [5], but the reality is that rerunning tests is still the main dynamic approach used to detect flakiness [12].

Pinto *et al*. [13] developed a prediction modelling approach that statically identifies flaky tests by analysing their code (test

code only). This approach is appealing compared to the current practice due to its static nature that a) does not require any test execution logging and analysis that is usually hard to implement on the fly, and usually are not supported by the test infrastructures, and b) the low overhead it entails, i.e., it reduces the execution cost caused by test reruns.

The original study (Pinto *et al*.) evaluated the performance of different machine learning models on different representations of the test code and found that the vocabulary of tests can predict test flakiness with 95% of accuracy (F1 score). Although encouraging, the original study was performed in a dataset covering one programming language (Java), evaluated in a time-insensitive manner and left open many additional questions related to the vocabulary of source code. Considering the importance of the problem we decided to perform further investigations. We believe that extensive and independent evaluations are also necessary to reach industrial adoption and practice.

Replication is essential to verify experimental results from previous studies. They are a key aspect of empirical software engineering as they bring evidence that observations made can hold (or not) under other conditions. Different types of replication exist [14], [15]. An *exact* replication attempts to reproduce the experiments following as closely as possible the initial procedures. By doing so, we learn that the first results were not caused by uncontrolled random factors. In *conceptual* replication, one or more dimensions can be changed to investigate to what extent the results hold.

In this paper, we present a conceptual replication of the study of Pinto *et al*. [13]. We start by considering a different validation methodology than the one used in the study of Pinto *et al*. [13]. We thus adopt a time-sensitive validation setting that better reflects the envisioned use case of the approach; at a given point in time, we train our predictor with historically identified flaky tests and inspect the model performance in predicting unseen flaky tests, i.e., with the subsequent "future" tests. We argue that this procedure is important to confirm the results and avoid biasing the predictions by considering future data.

Another aspect our study aims to evaluate is the generalisation of Pinto *et al*.'s findings, in particular to a different programming language. Therefore, we mine Python projects from GitHub and build a new dataset of 837 flaky tests. Then, we use it in order to evaluate the vocabulary-based flakiness prediction. This part of the analysis aims at re-validating the Pinto *et al*. findings on new and different data.

Finally, we go beyond the original study by considering an extended set of features. In particular, we attempt to predict flakiness using not only the vocabulary of test code (like Pinto *et al*. did) but also the Code Under Test (CUT). This endeavour follows the findings of many reports [1], [16], [17] revealing that much flakiness manifests in the CUT. Hence, we conduct a comparative study that highlights the impact of the two feature sets, *i.e.*, sources of vocabulary on flakiness prediction.

All in all, our results demonstrate that a more robust, time-sensitive validation has a consistent negative impact on the reported results of the original study (performance degrades by 7% on average) but, fortunately, do not invalidate the key conclusions of the study, *i.e.*, predictions are significantly better than random selections.

Additionally, we find re-assuring results that vocabulary-based models are more successful in Python than in Java (average performance of 80% in Python in contrast to 61% in Java), and perhaps surprisingly, that the information lying in the Code Under Test has a limited or no impact on the model performance. Taken together, these results corroborate the conclusion that the vocabulary of tests is indeed a viable and robust solution to the test flakiness problem.

II. The Pinto *et al*. Study

This work is a replication of the study by Pinto *et al*. [13]. In this section, we briefly summarise the approach they presented for flakiness prediction. We first present the dataset of existing flaky tests which they used in their study. Then, we explain their source code representation and prediction model. Finally, we recall their evaluation methodology and results.

A. Dataset

In their original study, Pinto *et al*. relied on the DeFlaker dataset, which was proposed by Bell *et al*. [18]. This dataset includes 1,874 flaky tests identified using the DeFlaker tool on multiple revisions of 24 open-source Java projects. Pinto *et al*. selected 1,403 flaky tests from this dataset to build their set of flaky tests. They also randomly selected tests that were not flagged as flaky by DeFlaker to form a set of *a priori* non-flaky tests. To mitigate the problem of class imbalance, both sets had the same size.

B. Prediction model

In order to prepare the classification inputs, Pinto *et al*. extracted identifiers that represent the test vocabulary and complexity. This extraction takes several steps. First, they localise the file where the test is defined. Then, they select all identifiers contained in this test, pre-process them by splitting them according to their camel-case syntax and converting them in lower-case. Finally, they remove stop words from the obtained set. Each flaky and non-flaky test is represented as follows:

- A vector of booleans indicating for each token if it is present in the test or not;
- The number of lines of code;
- The number of Java keywords contained in the test.

The last two features are used as a proxy for code complexity. The authors used these vectors as inputs for their prediction models. In particular, they evaluated the performance of five machine learning classifiers: Random Forest, Decision Tree, Naive Bayes, Support Vector Machine, and Nearest Neighbour.

C. Evaluation

1) Evaluation methodology: The authors follow a standard methodology to train and evaluate the five classifiers. That is, they split the whole set of test cases into a training set

containing 80% of the tests and a validation ("test") set containing the remaining 20%.

They report the standard precision, recall and F1-score metrics. The precision shows the proportion of correctly classified flaky tests. The recall shows the proportion of flaky tests found out of all existing ones. They focus their analysis on the F1-score, which combines precision and recall to assess the model performance. Detailed results for their different models are listed in Table I.

2) Results: Among the five trained models, the most promising one was Random Forest, having a performance as high as 0.95 for the F1-score. Altogether, the five models showed great performance on their dataset.

III. Replication Setup

Our key goal is to investigate whether the conclusions of Pinto *et al*. generalize to different flakiness scenarios, viz., (1) a time-sensitive prediction use case where flakiness information about past tests are used to predict flakiness in future (new) tests, (2) flakiness prediction in different programming languages, (3) the use of different sets of features involving both test code and code under test. Each of these scenarios gives rise to a research question that we answer in our study. In all scenarios, we use the model presented in the original study that gave the best performance. The model is based on a bag of words and a Random Forest of 100 trees i.e., the model which gave the best results in the original study. Our replication package containing code, models and datasets is available online[1].

A. Research Questions

We aim at answering the following research questions:

- **RQ1:** *How well do vocabulary-based models identify flaky tests when using a time-sensitive validation?*
- **RQ2:** *How well do vocabulary-based models identify flaky tests in other programming languages?*
- **RQ3:** *Is the vocabulary of Code Under Test useful for flakiness prediction?*

B. RQ1: How well do vocabulary-based models identify flaky tests when using a time-sensitive validation?

In the real world, one can picture different usages for a flaky test prediction model. For instance, in Continuous Integration (CI) environments where new changes (commits) making some tests fail are typically rejected, developers can ignore those failing tests that are likely to be flaky and isolate them for further investigation.

In another setting, the prediction model can also come as an IDE plugin hinting at tests that use keywords related to flakiness.

These scenarios illustrate the importance of the temporality of tests and code, as the model can be trained only on flaky tests detected previously to predict new occurrences. Moreover, the fact that the vocabulary of code changes as new commits are introduced makes it challenging for models trained on older data to predict flakiness in future code versions that are temporarily distant.

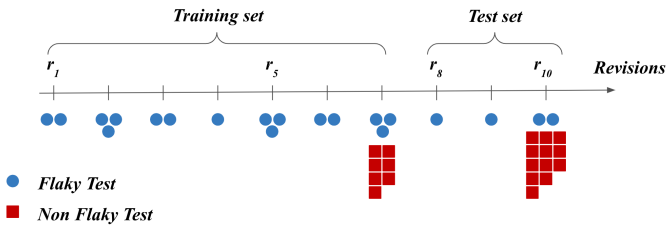

Fig. 1. Time-sensitive validation

The model can also be limited to flaky tests detected in one project, e.g., when the vocabulary linked to flakiness can differ from one project to another. Indeed, as reported in the literature [1], [3], different sources of flakiness exist such as concurrency issues, usage of date/time, I/O actions, API or network calls, etc. Thus, based on the project, the flakiness sources can differ and the vocabulary associated with it varies accordingly.

For all these reasons, we propose a novel, intra-project, time-sensitive setup for validating flakiness prediction models. This setup evaluates a model on its ability to predict new flaky tests with data that is assumed to be known from the past of the project.

To compare this setup with the one from Pinto *et al*., we rely on the DeFlaker dataset, which was also used in the original study. For each project, we select tests that were found flaky at any revision of the change history to form the Flaky Tests set FT. DeFlaker does not provide explicit information about tests that did not flake, as the tool can not guarantee that a test that did not fail (yet) is not flaky. We define Non Flaky Tests NFT as tests that were not found as flaky in any revision, that is, $NFT = T_{Total} - FT$.

Figure 1 explains how FT and NFT are selected in our time-sensitive validation. We split our dataset in order to have 80% of the FT from earlier revisions for our training set and 20% of the FT from "future" revisions for our test set.

We select the NFT from the revision where the last FT_{train} are selected for the training set and from the last revision where FT_{test} are selected for the test set.

To assess the impact of this new setup on model performance, we compare it with a classical setup where the model is trained and tested with flaky tests regardless of their observation date (i.e., the setup followed by Pinto *et al*.). In such setup, all flaky and non-flaky tests are grouped without accounting for their observation date. Then, the groups are randomly split into training and test sets following an 80/20 ratio.

To perform this comparison, we selected six projects from the DeFlaker dataset based on their numbers of flaky tests. These projects have at least 30 flaky tests, which we consider as a minimum necessary for training and testing a model. Table II presents these projects with their numbers of flaky and non-

[1]https://github.com/GuillaumeHaben/MSR2021-ReplicationPackage

TABLE I
MODEL PERFORMANCE OF THE PINTO *et al.* STUDY [13]

Algorithm	Precision	Recall	F1	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

TABLE II
DETAILS ABOUT THE JAVA PROJECTS USED IN OUR STUDY

Project	Earliest revision	latest revision	#FT	#NFT
achilles	2015-10-30	2016-09-05	51	392
hbase	2010-05-17	2010-06-21	98	120
okhttp	2014-03-06	2015-01-30	102	1178
oozie	2013-03-20	2013-05-31	1039	44
oryx	2015-01-06	2015-02-27	38	286
togglz	2016-01-23	2016-06-17	20	256

flaky tests. We also present the dates of the first and last flaky tests identified in these projects. We split this dataset according to the two validation setups, then we build our prediction model, train it and contrast the results of both setups.

C. RQ2: How well do vocabulary-based models identify flaky tests in other programming languages?

1) Predicting flaky tests in Python: Another goal of our study is to evaluate the generalisability of the original study to other programming languages. For this purpose, we propose to assess the performance of flakiness prediction models on Python projects. We chose Python because it is the most popular language used in modern projects and it is commonly used for machine learning, web development, game development, and many other applications.

Python comes with its set of testing frameworks. We focus our study on Pytest [19]. Pytest is the equivalent of Junit for Python and enables developers to write tests for their programs. It is one of the main testing frameworks used in the open-source community and in the industry. Pytest comes with its lot of features and plugins. Especially, a specific module to handle flaky tests can be used with Pytest: flaky[2]. This module allows developers to annotate tests as flaky to automatically rerun them in case of failure. The developer can also configure the maximum amount of reruns to attempt and the minimum number of passes required. This annotation can be added to a test function or directly to the test class, giving its property to all of its tests. Figure 2 shows an example of a test marked as *@flaky* taken from the Typed_python project[3].

We mined GitHub using the source-graph API[4], searching for Python projects containing the annotation *@flaky*. This

[2]https://pypi.org/project/flaky/
[3]https://github.com/APrioriInvestments/typed_python
[4]https://sourcegraph.com/search

```
75
76      @flaky(max_runs=3, min_passes=1)
77      def test_sort_perf_simple(self):
78          x = ListOf(float)(numpy.random.uniform(size=1000000))
79
80          sorting.sorted(x[:10])
81
82          t0 = time.time()
83          sorting.sorted(x)
84          t1 = time.time()
85          sorted(x)
86          t2 = time.time()
87
88          speedup = (t2 - t1) / (t1 - t0)
89
90          # I get about 3
91          self.assertGreater(speedup, 1.5)
92
```

Fig. 2. Example of a test labelled @flaky

TABLE III
PYTHON PROJECTS USED IN OUR STUDY

Project	SHA	#FT	#NFT
bokeh	ddc22b8	100	2505
cassandra-dtest	8cb6bd2	72	4221
celery	0833a27	54	2890
jira	7fa3a45	131	59
pipenv	8e64873	32	1612
python-amazon	84c16f5	35	15
python-telegram-bot	8e7c0d6	186	1382
spyder	413c994	173	1086
typed-python	96e7ebd	54	6034

process yielded 110 projects with a total of 1,304 tests marked as flaky. Similarly to our first experimentation, we only select projects in which we have enough flaky tests to train and test a model, *i.e.* 30 flaky tests. This results in a dataset of 9 projects and 837 tests marked by developers as flaky. Table III shows these projects with their number of flaky and non-flaky tests. Compared to the Java dataset, we were able to obtain more projects with more flaky tests for our study. To the best of our knowledge, this is the first dataset of flaky tests in Python.

It is worth noting that in this research question, we evaluate the performance of the model to predict flaky tests in a single revision. Therefore, we reuse the typical 80/20 dataset split as followed by Pinto *et al.*. That is, we are rather focusing

TABLE IV
CLASSIFIER PERFORMANCE FOR PYTHON PROJECTS WITH MANIFEST FLAKY TESTS

Project	#reruns	#@flaky	#manifest FT
bokeh	200	100	1
celery	300	54	2
python-telegram-bot	300	186	20

on confirming that the approach works as well in Python and that a model can learn features differentiating tests labelled as *@flaky* from the ones that are not. To extract these features, we use a bag of words representation of the test, as in Java. We also carefully remove the *@flaky* annotations, as keeping it in the vocabulary would bias our model towards recognising this annotation rather than the code vocabulary.

2) Predicting manifest flaky tests: We perform further analysis in Python to assess the usefulness of a vocabulary-based model. Our objective is to evaluate the ability of a model to identify *manifest* flaky tests based on training with tests labelled as flaky by developers. We consider as manifest flaky, every test for which we are able to observe non-deterministic behaviour dynamically. This means that the test fails and passes at least once after several reruns. To identify these manifest flaky tests, we reran 200 to 300 times the test suite of the three projects Bokeh, Celery and Python-telegram-bot. We run the test suites on a Mac machine with a 2,4 GHz 8-Core i9 processor and 32Gb of RAM. The results of these reruns are presented in the table IV. The column #@flaky shows the number of tests labelled as flaky in each project.

We observe that despite the high number of reruns (800), only 23 tests have a flaky behaviour. This outcome is not surprising as flaky tests are, by nature, difficult to reproduce. To assess the model performance in detecting manifest flaky tests, we focus on the only project that has a reasonable amount of manifest flaky tests, namely Python-telegram-bot. We use the 20 manifest flaky tests found during our reruns as a test set, completed by 20 randomly selected tests that are not labelled as flaky. For the training set, we use the flaky and non-flaky tests minus the tests present in the test set.

D. RQ3: Is the vocabulary of Code Under Test useful for flakiness prediction?

So far, the flakiness prediction is only based on features taken from the test code. However, flaky tests can be due to infrastructure or environmental issues (*e.g.* lack of available resources in the CI, service or network unavailable, etc), to the test itself (*e.g.* usage of dates, randomness, order dependency, etc), or to the CUT (*e.g.* non-determinism, concurrency, etc). Notably, Luo *et al*. [1] showed that 24% of the fixes for flaky tests were applied to the CUT and that among them, 94% fixed a bug in the CUT. Hence, it can be judicious to consider information from the CUT in flakiness prediction models. We propose to extend the original study by including the vocabulary of the CUT in test representation.

The main issue when considering the CUT is that computing the code coverage of each test during each revision would bring significant overheads. Besides, retrieving the exact code coverage dynamically goes against the goal of the static prediction, which is to reduce dynamic costs. To avoid this overhead, we propose a lightweight approach that relies on Information Retrieval (IR) to estimate the CUT.

IR techniques have been used to solve different software engineering problems [20]–[22]. IR aims at quickly and automatically retrieving relevant information among a set of documents based on keywords taken from a user query. In our case, the query is the tokens of a test and the set of documents is the set of all functions (or methods) defined in the project. Our hypothesis is that functions from the CUT of a test are likely to use similar keywords (*i.e.* variable names, API calls, etc) as the test. We are then looking for the most similar functions to our test function. To do so, we use a cosine similarity between a test case and a function from the CUT. Cosine similarity is defined with:

$$cosSimilarity = \cos(Tc, Func) = \frac{Tc \cdot Func}{|Tc||Func|}$$

where Tc is the vector representing the test code and $Func$ is the vector representing the function code. The result of a cosine similarity ranges from -1, meaning that the query - our test case - is completely different from the document - our function - to 1, where the query is perfectly similar to the document. In our case, we select the top three most similar functions for each test.

Algorithm 1: Cost effective retrieval of the CUT

```
Inputs:
Test[]
Function[]
Outputs:
TestWithCUT[]
Procedure CUT_SELECTION(Test[], Function[])
foreach test T ∈ Test[] do
    similarityMeasures[]
    Tvector = transform(T)
    foreach function F ∈ Function[] do
        fit(T + F)
        Fvector = transform(F)
        cosTF = cosSimilarity(Tvector, Fvector)
        similarityMeasures.Append(cosTF)
    end
    similarityMeasures.Sort()
    similarityMeasures.Slice(0, 2)
    T.append(similarityMeasures)
    TestWithCUT.append(T)
end
return TestWithCUT[]
```

Algorithm 1 describes the process of associating the CUT to each test. In order to compute the cosine similarity between the test and a function, we use the Text tokenisation utility

class from the Keras library[5]. We first fit the *Tokeniser* with the vocabulary from all tests and functions. Then, we transform the text of test and function bodies by creating a vector for each one of them of a length equal to the size of the vocabulary. In this vector, each element represents the number of times a word appears in the body. After extracting the vectors, we compute the cosine similarity between the current test and all functions and store results. We finally filter to only keep functions that have a high score, *i.e.* that they are the closest to the test. The body representation of the selected functions is used as a new set of features for the flakiness prediction model.

IV. Results

A. RQ1: How well do vocabulary-based models identify flaky tests when using a time-sensitive validation?

In this research question, we compare prediction model performance using a time-sensitive validation and a classical validation. The accuracy of a model is sensitive to the class imbalance. In particular, the precision and recall metrics can easily be impacted when one class is under (or over) represented. To alleviate this issue, we also report the Matthews Correlation Coefficient (MCC). In our case, we focus on the MCC as it takes into consideration all four entries of the confusion matrix, True Positives (TP), True Negatives (TN), False Positives (FP) and False Negatives (FN). MCC is given by:

$$\frac{TN \times TP - FP \times FN}{\sqrt{(TN+FN)(TP+FP)(TN+FP)(FN+TP)}}$$

The MCC score ranges from 1, where the classifier did a perfect job (FP = FN = 0) to -1, where the classifier always predicted the wrong class (TP = TN = 0). An MCC value of 0 would indicate that the model is no better than a random guess.

Figures 3-5 show the performance of our Random Forest classifier under time-sensitive and classical validation. Overall, we observe that the validation setup has an impact on the classifier performance. This impact varies significantly depending on the project, its size, and history of flaky tests. The projects Achilles, Hbase, OkHttp, and Togglz observe a decrease in their MCC score. The largest performance drop is observed in the OkHttp project, where the MCC dropped from 39% to 18%. The two exceptions are for Oozie and Oryx, where MCC increased respectively by 10% and 13%. In the case of Oryx, this can be explained by the fact that most of the flaky tests come from one revision, thus, the time-sensitive validation has little to no impact. The difference can then be explained by the random selection of the samples when splitting the training and test set. The phenomenon is only present for this project. In the case of Oozie, there is a considerable imbalance between the number of flaky tests (1039) and non-flaky tests (44). Hence, the test set contains only 9 non-flaky tests, which might not be enough to draw conclusions.

[5]https://keras.io/api/preprocessing/text/

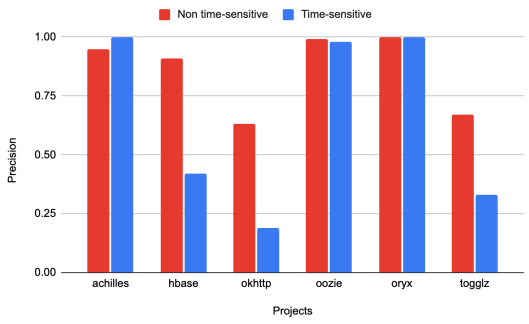

Fig. 3. Precision under classical and time-sensitive validations.

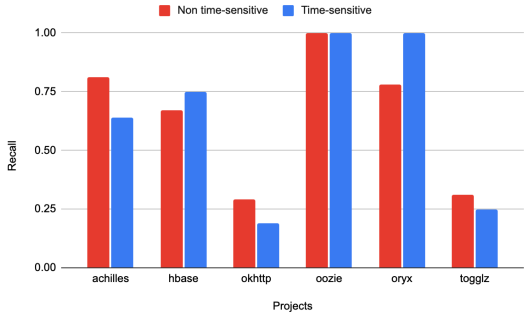

Fig. 4. Recall under classical and time-sensitive validations.

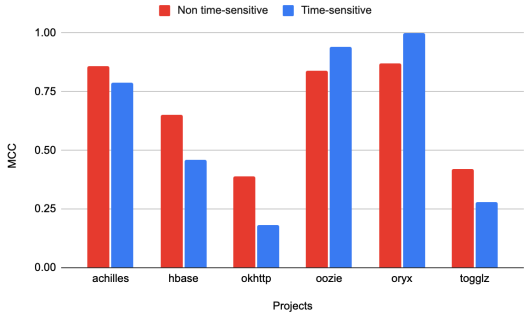

Fig. 5. MCC under classical and time-sensitive validations.

> The performances of a vocabulary-based model decrease under a time-sensitive validation (up to 21% drop for MCC). Nonetheless, the approach is still able to decently predict flaky tests.

B. RQ2: How well do vocabulary-based models identify flaky tests in other programming languages?

1) Predicting flaky tests in Python: Table V reports on the model performance when predicting flaky tests in 9 Python projects.

First, we observe that for 5 projects out of 9, the model reaches a great performance with MCC and F1 scores greater

TABLE V
CLASSIFIER PERFORMANCE FOR PYTHON PROJECTS

Project	Precision	Recall	F1	MCC	AUC
bokeh	**1.00**	**0.91**	**0.95**	**0.95**	**0.95**
cassandra-dtest	**0.96**	0.43	0.58	0.63	0.71
celery	0.85	0.54	0.64	0.66	0.77
jira	**0.98**	**0.99**	**0.99**	**0.95**	**0.98**
pipenv	0.78	0.19	0.30	0.37	0.60
python-amazon	**0.97**	**1.00**	**0.99**	**0.95**	**0.96**
python-telegram-bot	**1.00**	**0.99**	**1.00**	**0.99**	**1.00**
spyder	**0.92**	0.77	0.83	0.82	0.88
typed-python	**1.00**	0.86	**0.91**	**0.92**	**0.93**

TABLE VI
CLASSIFIER PERFORMANCE FOR MANIFEST FLAKY TEST IN THE PYTHON-TELEGRAM-BOT PROJECT

Project	Precision	Recall	F1	MCC	AUC
python-telegram-bot	1.00	1.00	1.00	1.00	1.00

than 90%. For the rest of the projects, these scores are always higher than 50%, except for Pipenv, which shows the lowest results with an MCC of 37%. Similarly, all the studied projects have a precision greater than 78% and 7 out of the 9 studied projects have a precision higher than 90%. These observations show that the vocabulary-based model is able to predict flaky tests with decent performance in Python projects.

2) Predicting manifest flaky tests: Table VI shows the model performance in detecting manifest flaky tests based on tests marked as flaky by developers. The results show a perfect performance with a MCC and F1-score values of 100%, confirming that a model trained on tests labelled by developers can be used to predict manifest flaky tests. Interestingly, 2 of the 20 manifest tests were not labelled as flaky by the developers and were only identified with the reruns. Yet, the model was able to predict them by only learning from tests marked by developers. In a real-world scenario, we could picture the model finding those tests and automatically annotating them.

Figure 6 shows the test `test_idle()` from the class `TestUpdater`[6]. Over 300 reruns, this test failed intermittently because of a concurrency issue where a scheduler has been shut down. Indeed, the test body contains several keywords related to time and concurrency, which are common causes of flakiness, *e.g.* `Thread`, `sleep`, `idle`. In order to understand how the model predicted that this test is flaky, we analyse the most important features of the model. These features do not completely reflect the model prediction and they can be biased [23], but they give us an idea of the vocabulary that the classifier is using for its predictions. In the project Python-telegram-bot, we found that the top ten features include the keywords: `process`, `timeout`, `duration`, `seconds`, which are also related to time and concurrency. Hence, the model's ability to predict the test flakiness based on the vocabulary.

[6]https://github.com/python-telegram-bot/python-telegram-bot

```
@signalskip
def test_idle(self, updater, caplog):
    updater.start_polling(0.01)
    Thread(target=partial(self.signal_sender, updater=updater)).start()

    with caplog.at_level(logging.INFO):
        updater.idle()

    rec = caplog.records[-2]
    assert rec.getMessage().startswith('Received signal {}'.format(signal.SIGTERM))
    assert rec.levelname == 'INFO'

    rec = caplog.records[-1]
    assert rec.getMessage().startswith('Scheduler has been shut down')
    assert rec.levelname == 'INFO'

    # If we get this far, idle() ran through
    sleep(0.5)
    assert updater.running is False
```

Fig. 6. A manifest flaky test not labelled @flaky

Figure 7 shows the test `test_to_dict()` from the class `TestStickerSet`, which is also manifestly flaky but the developers did not mark it as such. Unordered collections have been identified as a cause of flakiness by several works as developers can wrongly assume that elements of a collection will be returned in a specific order [1], [24]. In Python, the return order of dictionaries has varied over the different versions [25], [26]. In our case, we found that the keyword `dict`, which is present in a large number in this test, was among the first eight most important features of our classifier. This feature allowed the vocabulary-based model to predict that this test is flaky.

```
    def test_to_dict(self, sticker):
        sticker_dict = sticker.to_dict()

        assert isinstance(sticker_dict, dict)
        assert sticker_dict['file_id'] == sticker.file_id
        assert sticker_dict['file_unique_id'] == sticker.file_unique_id
        assert sticker_dict['width'] == sticker.width
        assert sticker_dict['height'] == sticker.height
        assert sticker_dict['is_animated'] == sticker.is_animated
        assert sticker_dict['file_size'] == sticker.file_size
        assert sticker_dict['thumb'] == sticker.thumb.to_dict()
```

Fig. 7. A manifest flaky test not labelled @flaky

We conclude that the approach is extendable to the Python language, supporting the idea that the vocabulary-based prediction can be generalisable to other projects and programming languages. Moreover, we saw that we can take advantage of a flaky tests classifier using vocabulary-based features in order to identify vocabulary linked to flakiness and help developers write better quality tests.

> Vocabulary-based models can be generalised to other projects and programming languages. Besides, these models can leverage annotated flaky tests to predict and annotate manifest flaky tests that were not known to developers.

C. RQ3: Is the vocabulary of Code Under Test useful for flakiness prediction?

Figures 8-10 show the results of our prediction model in Java projects, while figures 11-13 present the model performance in Python projects.

Fig. 8. Precision score in Java projects

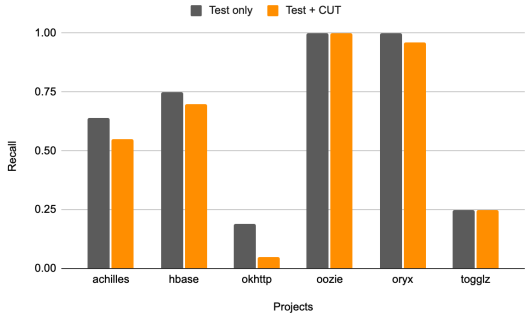

Fig. 9. Recall score in Java projects

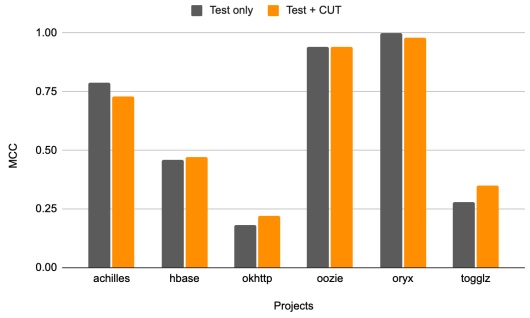

Fig. 10. MCC score in Java projects

In figures 8-10, we observe that the impact of including the CUT does not have a consistent impact on the model performance in Java projects. Adding the CUT improves the model performance in Hbase, Okhttp and Togglz, with an increase of the MCC score between 1% and 7%. However, the opposite effect is observed in the projects Achilles and Oryx where the MCC dropped by 6% and 2% respectively. As for the Oozie project, including the CUT does not seem to impact the model performance. Nonetheless, these performance improvements and losses remain minor in all the studied Java projects.

Figures 11-13 show a similar effect of the CUT usage in Python projects. Out of the nine studied, six projects report

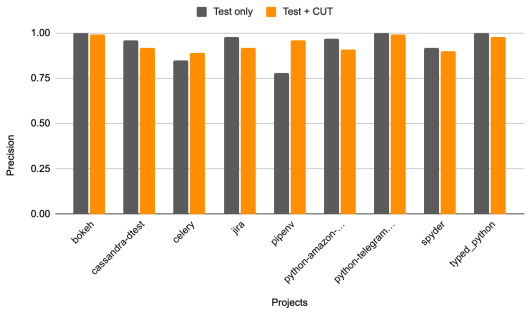

Fig. 11. Precision score in Python projects

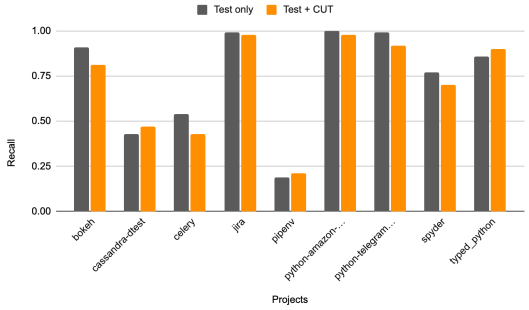

Fig. 12. Recall score in Python projects

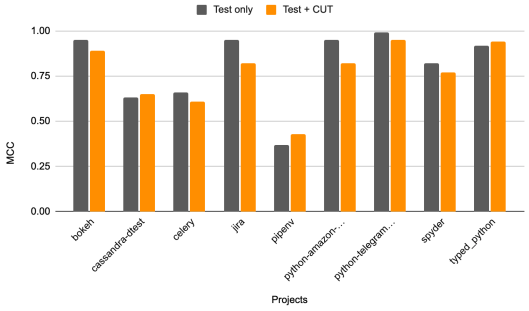

Fig. 13. MCC score in Python projects

a lower performance when adding the CUT to the features. This performance loss is up to 13% in the projects Jira and Python-amazon. On the other hand, the projects Cassandra-dtest, Pipenv, and Typed_python have better predictions when the CUT is used — an increase in the MCC value by 2%, 6% and 2% respectively. Based on the observations in both Java and Python, we conclude that including the CUT does not consistently improve the performance of a vocabulary-based model for predicting flaky tests.

> Surprisingly, the vocabulary of the Code Under Test, which is commonly considered as a source of flakiness, does not improve the performance of flakiness prediction models.

V. Threats to validity

A. Construct Validity

One possible threat to the study's construct validity is our choice and selection of flaky tests in Python. It is possible that tests that are marked as flaky by developers are not actually flaky. In particular, developers could abuse of the annotation and mark non-flaky tests to forecast flaky behaviour. To inspect this point, we manually analysed projects from our dataset to check if this behaviour is prevalent. We found that some projects (like Jira and Python-telegram-bot) use the annotations to mark all class tests as flaky. However, this usage seems judicious as the class tests performed GUI testing, which is known for being a major cause of flakiness. Moreover, an abusive usage of this annotation by developers seems unlikely considering the rerun costs. When running the test suites, we observed that one pass can take a long time. Hence, it is not in the best interest of developers to mark as many tests as flaky to anticipate flakiness as this would largely increase the execution time as soon as there are test failures. We believe that the usage of annotated flaky tests in our study is reasonable given the lack of large datasets of flaky tests, especially for programming languages other than Java. Ideally, the annotated flaky tests would be validated by rerunning them and exhibiting their non-deterministic behaviour. Nevertheless, the reproduction remains very challenging for flaky tests in general and even tests identified in other datasets are hardly reproducible [13], [27].

Another threat to construct validity could be our approach for retrieving the CUT. Intending to design a fast and lightweight approach, we used Information Retrieval to estimate the real code coverage of each test. This approximation can be responsible for the noise brought in the features. To investigate this point, we assessed the CUT effect when using other retrieval approaches. First, we retrieved an approximation of the CUT by using Static Call Graph. We selected all functions called by the test as the CUT and we do not explore what those functions call. Even if this approach only includes a subset of the CUT and flakiness can be caused by functions deeper in the call graph, we believe that keywords in the top functions should serve as proxy. Results for this approach were similar to the ones presented in RQ3. The performance scores slightly decrease or increase from one project to another without showing a significant impact on the prediction performance. Furthermore, we computed the code coverage for projects where we managed to build and run the test suite. This task is challenging, especially in Java where the flaky revisions are from several years ago and dependencies are easily missing from central repositories. We successfully retrieved the real code coverage for revisions of Togglz and Oryx using the GZoltar tool [28]. This tool allows us to get a coverage matrix representing each line covered by the test case. We used this matrix to retrieve the exact CUT and include it as a feature for our prediction model. For both projects, the CUT inclusion had an impact on the model performance, which is very similar to the one observed with the CUT retrieved with IR. Hence, we believe that the results observed in RQ3 are not flawed by the CUT retrieval.

B. Internal Validity

One possible threat to internal validity is the definition of non-flaky tests. The datasets that we used for both Java and Python, only mark flaky tests and do not provide information about non-flaky tests. Consequently, we considered all tests that were not marked as flaky to be non-flaky. Yet, some of these tests can be flaky even though DeFlaker or the developer did not mark them as such. This limitation is not unique to our study as it is theoretically impossible to prove that a test is not flaky. To the best of our knowledge, there are no datasets, neither in formal nor in grey literature, that mark explicitly non-flaky tests. On top of that, our study results show that there is a clear distinction between the classes of flaky and non-flaky tests. Accordingly, it is unlikely that a significant fraction of the non-flaky tests is actually flaky.

One common threat to the internal validity of replication studies is potential errors in the reproduction (*e.g.* settings and library usage). To alleviate this threat, we carefully examined the GitHub repository of the original work [29] to understand and reproduce their implementation details. Besides, the goal of our study is not to exactly reproduce the original work and our results align well with the original findings.

C. External Validity

The main threat to our external validity is the size and nature of our datasets. For Java, we relied on the DeFlaker dataset since it is the largest open-source set of flaky tests and it has already been used in many flakiness studies [13], [30]. As for Python, we built a dataset of 837 flaky tests from 9 projects by mining GitHub repositories. For the sake of generalisability, it would have been preferable to include more projects and flaky tests. Nevertheless, our intra-project setting required a minimum number of flaky tests per project and limited our choices. We encourage future studies to replicate this study on larger datasets, including industrial projects.

VI. Related Work

Studies on flakiness. Luo *et al.* [1] conducted the first extensive study that aims to understand the root causes of flakiness. They analysed 201 commits from 51 open source projects to explore the fixing strategies adopted by developers. Their analysis showed that the top categories for flakiness are async wait, concurrency, and test order dependency. Lam *et al.* [17] also carried a large-scale study on flaky tests, attempting to understand when tests first become flaky. They ran detectors on 55 Java projects and found that 75% of the 245 detected flaky tests were flaky as soon as introduced in the test suite, indicating potential benefits for developers to run flaky-tests detectors when adding new tests. Eck *et al.* [31] conducted a survey with 121 professional developers. They found that flakiness is indeed perceived as an important issue by developers as it wastes developers time and breaks the trust in test suites. They also report that test flakiness can be

caused by the code under test. Researchers also investigated flakiness in different application domains. For instance, Thorve *et al*. [16] reproduced the study of Luo *et al*. for Android applications and identified new flakiness root causes. Other studies [32]–[34] measured the impact of flakiness on mutation testing and automated program repair. They showed that this impact can be significant.

Industrial studies. Several studies have been carried out in the industry, showing the need for new solutions to reduce the flakiness impact. Leong et al. [4] studied the effects of test flakiness in Google regression testing practices, test selection in particular, and reported that more than 80% of test transitions are due to flaky tests. The study also investigated the impact of test flakiness when evaluating regression testing techniques and found that flakiness significantly overestimate evaluations' performance. Lam *et al*. [3] conducted a study about flaky tests in Microsoft projects and show that even if the number of flaky tests in a project is low, the number of build failures caused by flakiness can still be significant. They also presented a tool, RootFinder, which attempts to mitigate flaky tests by analysing differences in their logs. A follow-up study by Lam *et al*. [35], still at Microsoft, proposed an automated solution, FaTB, in order to decrease the runtime of their test suite by modifying tests timeouts and waits without changing their flake rate. Kowalczyk *et al*. [5] were able to reduce flakiness by 44% by implementing a flakiness scoring system at Apple.

Tools & Datasets. Other works introduced frameworks for flaky test detection alongside their dataset. iDFlakies [36] can detect flaky tests by rerunning test suites several times in different orders. Using iDFlakies, the authors built a dataset of 422 order-dependent tests in Java projects. Shi *et al*. [37] presented iFixFlakies, a framework to automatically patch order dependent flaky tests using delta-debugging. Silva *et al*. [38] proposed SHAKER, a technique to reveal flaky tests in a more efficient way than reruns by adding noise in the environment. Dutta *et al*. [24] presented a technique, FLASH, to detect flaky tests caused by random seeds used in probabilistic programming systems and machine learning frameworks. DeFlaker [18] finds tests that are likely to be flaky if they happen to fail without having executed any changed code. DeFlaker was used to build a dataset of 1,874 flaky tests from 24 open-source projects. We partially rely on this dataset in our study of Java projects.

Flakiness prediction. A few works explored the possibility of using machine learning to predict flakiness based on training data. King *et al*. [39] presented a bayesian network model for classifying and predicting flaky tests. In their work, they view the test flakiness as a disease with its symptoms and common effects. They built their model on top of factors that can cause flakiness (*e.g.* high assertion count, explicit wait count, high cyclomatic complexity) and train their bayesian network classifier on historical data from the Ultimate Software company. They presented a flakiness prediction accuracy of 65.7%. Bertolino *et al*. [30] presented FLAST, a static prediction model based on K-nearest neighbours that aims to identify similar (flaky) tests. Finally, Pinto *et al*. [13] presented the study that we replicate in this paper. They presented a bag of words approach for representing tests and trained five different machine learning models to evaluate their performance. They show that detecting flaky tests with reruns is challenging. To do so, they relied on the data of all DeFlaker projects, whereas in our replication, we focus on an intra-project analysis. They also question the impact of different features used for their models like the use of stemming, lower-casing, splitting, stop word removal, in the goal of optimising their classifier performance. In comparison, we also explore additional features by exploring the effect of including the CUT on the performance of the prediction model. Finally, they examine the vocabulary that is associated with flakiness in order to show that their approach can give hints to developers based on some keywords. Their study was done using Java projects, whereas we performed our study in Java and Python to increase the confidence in the generalisation of the approach.

VII. Conclusion and Future Work

This paper explored the usability and performance of vocabulary-based models in predicting flaky tests. We presented a conceptual replication of the study of Pinto *et al*., following three axes.

- First, we evaluated the prediction performances under a time-sensitive validation setting that better reflects the envisioned use case for the approach. We found that a more robust validation has a consistent negative impact on the reported results of the original study (performance degrades by 7% on average). Fortunately, this performance degradation does not invalidate the key conclusions of the study as the model predictions are significantly better than random selections.
- Second, we evaluated the generalisability of a vocabulary-based model to other programming languages. We found re-assuring results that vocabulary-based models are more successful in Python than in Java (average performance of 80% in Python in contrast to 61% in Java). We also showed that these models can leverage flaky tests annotated by developers to predict and annotate manifest flaky tests that were not known to developers.
- Third, we conducted a comparative study that highlights the impact of features lying in the CUT on the prediction performance. Surprisingly, we found that the vocabulary of the CUT, which is commonly considered as a source of flakiness, does not improve the performance of vocabulary-based models.

On top of these findings, this paper presents a new large dataset of flaky tests mined from developer annotations in Python projects on GitHub. This dataset and our experiment toolset are available in a comprehensible replication package.

Acknowledgements

This work is supported by the Facebook 2019 Testing and Verification research awards and PayPal.

REFERENCES

[1] Q. Luo, F. Hariri, L. Eloussi, and D. Marinov, "An empirical analysis of flaky tests," in *Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, vol. 16-21-November-2014. Association for Computing Machinery, nov 2014, pp. 643–653.

[2] M. Rehkopf, "What is continuous integration — atlassian," https://www.atlassian.com/continuous-delivery/continuous-integration, (Accessed on 01/12/2021).

[3] W. Lam, P. Godefroid, S. Nath, A. Santhiar, and S. Thummalapenta, "Root Causing Flaky Tests in a Large-Scale Industrial Setting," in *Proceedings ofthe 28th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '19)*. Beijing, China: ACM Press, 2019, pp. 101–111.

[4] C. Leong, A. Singh, M. Papadakis, Y. L. Traon, and J. Micco, "Assessing transition-based test selection algorithms at google," in *Proceedings of the 41st International Conference on Software Engineering: Software Engineering in Practice, ICSE (SEIP) 2019, Montreal, QC, Canada, May 25-31, 2019*, H. Sharp and M. Whalen, Eds. IEEE / ACM, 2019, pp. 101–110. [Online]. Available: https://doi.org/10.1109/ICSE-SEIP.2019.00019

[5] E. Kowalczyk, K. Nair, Z. Gao, L. Silberstein, T. Long, and A. Memon, "Modeling and ranking flaky tests at apple," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Software Engineering in Practice*, ser. ICSE-SEIP '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 110–119. [Online]. Available: https://doi.org/10.1145/3377813.3381370

[6] J. Micco, "The State of Continuous Integration Testing Google," 2017.

[7] H. Jiang, X. Li, Z. Yang, and J. Xuan, "What causes my test alarm? automatic cause analysis for test alarms in system and integration testing," in *Proceedings of the 39th International Conference on Software Engineering*, ser. ICSE '17. IEEE Press, 2017, p. 712–723. [Online]. Available: https://doi.org/10.1109/ICSE.2017.71

[8] M. contributors, "Test verification - mozilla — mdn," https://developer.mozilla.org/en-US/docs/Mozilla/QA/Test_Verification, March 2019, (Accessed on 01/12/2021).

[9] M. Harman and P. O'Hearn, "From Start-ups to Scale-ups: Opportunities and Open Problems for Static and Dynamic Program Analysis," in *2018 IEEE 18th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE, sep 2018, pp. 1–23. [Online]. Available: https://ieeexplore.ieee.org/document/8530713/

[10] J. Palmer, "Test flakiness – methods for identifying and dealing with flaky tests : Spotify engineering," https://engineering.atspotify.com/2019/11/18/test-flakiness-methods-for-identifying-and-dealing-with-flaky-tests/, November 2019, (Accessed on 01/12/2021).

[11] A. Micco, John & Memon, "Gtac 2016: How flaky tests in continuous integration - youtube," https://www.youtube.com/watch?v=CrzpkF1-VsA, December 2016, (Accessed on 01/12/2021).

[12] J. Listfield, "Google testing blog: Where do our flaky tests come from?" https://testing.googleblog.com/2017/04/where-do-our-flaky-tests-come-from.html, April 2017, (Accessed on 01/12/2021).

[13] G. Pinto, B. Miranda, S. Dissanayake, M. D'Amorim, C. Treude, and A. Bertolino, "What is the Vocabulary of Flaky Tests?" *Proceedings - 2020 IEEE/ACM 17th International Conference on Mining Software Repositories, MSR 2020*, pp. 492–502, 2020.

[14] F. J. Shull, J. C. Carver, S. Vegas, and N. Juristo, "The role of replications in Empirical Software Engineering," *Empirical Software Engineering*, vol. 13, no. 2, pp. 211–218, 2008.

[15] O. S. Gómez, N. Juristo, and S. Vegas, "Understanding replication of experiments in software engineering: A classification," *Information and Software Technology*, vol. 56, no. 8, pp. 1033–1048, 2014.

[16] S. Thorve, C. Sreshtha, and N. Meng, "An empirical study of flaky tests in android apps," *Proceedings - 2018 IEEE International Conference on Software Maintenance and Evolution, ICSME 2018*, pp. 534–538, 2018.

[17] W. Lam, S. Winter, A. Wei, T. Xie, D. Marinov, and J. Bell, "A large-scale longitudinal study of flaky tests," *Proceedings of the ACM on Programming Languages*, vol. 4, no. OOPSLA, pp. 1–29, 2020.

[18] J. Bell, O. Legunsen, M. Hilton, L. Eloussi, T. Yung, and D. Marinov, "DeFlaker: Automatically Detecting Flaky Tests," in *Proceedings of the 40th International Conference on Software Engineering - ICSE '18*. New York, New York, USA: ACM Press, 2018, pp. 433–444. [Online]. Available: http://dl.acm.org/citation.cfm?doid=3180155.3180164

[19] "Pytest documentation," https://docs.pytest.org/en/stable/, (Accessed on 03/18/2021).

[20] R. K. Saha, L. Zhang, S. Khurshid, and D. E. Perry, "REPiR : An Information Retrieval based Approach for Regression Test Prioritization," *FSE '14, Hongkong*, 2014.

[21] F. Palomba, A. Zaidman, and A. De Lucia, "Automatic test smell detection using information retrieval techniques," *Proceedings - 2018 IEEE International Conference on Software Maintenance and Evolution, ICSME 2018*, pp. 311–322, 2018.

[22] M. Azizi and H. Do, "ReTEST: A Cost Effective Test Case Selection Technique for Modern Software Development," *Proceedings - International Symposium on Software Reliability Engineering, ISSRE*, vol. 2018-October, pp. 144–154, 2018.

[23] "Permutation importance vs random forest feature importance (mdi) - scikit-learn 0.24.0 documentation," https://scikit-learn.org/stable/auto_examples//inspection/plot_permutation_importance.html, (Accessed on 01/12/2021).

[24] S. Dutta, A. Shi, R. Choudhary, Z. Zhang, A. Jain, and S. Misailovic, "Detecting flaky tests in probabilistic and machine learning applications," *ISSTA 2020 - Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pp. 211–224, 2020.

[25] P. S. Foundation, "What's new in python 3.1 - python 3.9.1 documentation," https://docs.python.org/3/whatsnew/3.1.html#pep-372-ordered-dictionaries, March 2021, (Accessed on 01/12/2021).

[26] A. Barnert, "python - how should i test that dictionaries will always be in the same order? - stack overflow," https://stackoverflow.com/questions/50475966/how-should-i-test-that-dictionaries-will-always-be-in-the-same-order/50476093#50476093, May 2018, (Accessed on 01/12/2021).

[27] W. Lam, S. Winter, A. Astorga, V. Stodden, and D. Marinov, "Understanding Reproducibility and Characteristics of Flaky Tests Through Test Reruns in Java Projects," pp. 403–413, 2020.

[28] (2020) Gzoltar - automatic testing & debugging using spectrum-based fault localization (sfl). https://gzoltar.com/. (Accessed on 01/11/2021).

[29] M. d'Amorim. (2020, April) damorimrg/msr4flakiness. https://github.com/damorimRG/msr4flakiness/. (Accessed on 01/11/2021).

[30] A. Bertolino, E. Cruciani, B. Miranda, and R. Verdecchia, "Know Your Neighbor: Fast Static Prediction of Test Flakiness," *Proceedings of the International Conference on Software Engineering (ICSE)*, 2020. [Online]. Available: https://ieeexplore.ieee.org

[31] M. Eck, M. Castelluccio, F. Palomba, and A. Bacchelli, "Understanding Flaky Tests: The Developer's Perspective," *arXiv*, pp. 830–840, 2019.

[32] M. Cordy, M. Papadakis, R. Rwemalika, and M. Harman, "FlakiMe: Laboratory-controlled test flakiness impact assessment. A case study on mutation testing and automated program repair," *arXiv*, 2019.

[33] A. Shi, J. Bell, and D. Marinov, "Mitigating the effects of flaky tests on mutation testing," *ISSTA 2019 - Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pp. 296–306, 2019.

[34] Y. Qin, S. Wang, K. Liu, and X. Mao, "On the Impact of Flaky Tests in Automated Program Repair," no. February, 2021.

[35] W. Lam, K. Muslu, H. Sajnani, and S. Thummalapenta, "A study on the lifecycle of flaky tests," *Proceedings - International Conference on Software Engineering*, pp. 1471–1482, 2020.

[36] W. Lam, R. Oei, A. Shi, D. Marinov, and T. Xie, "IDFlakies: A framework for detecting and partially classifying flaky tests," *Proceedings - 2019 IEEE 12th International Conference on Software Testing, Verification and Validation, ICST 2019*, pp. 312–322, 2019.

[37] 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 European Software Engineering Conference and Symposium on the Foundations ofSoftware Engineering (ESEC/FSE '19)*, 2019.

[38] D. Silva, L. Teixeira, and M. D'Amorim, "Shake It! Detecting Flaky Tests Caused by Concurrency with Shaker," *Proceedings - 2020 IEEE International Conference on Software Maintenance and Evolution, ICSME 2020*, pp. 301–311, 2020.

[39] T. M. King, D. Santiago, J. Phillips, and P. J. Clarke, "Towards a Bayesian Network Model for Predicting Flaky Automated Tests," *2018 IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C)*, pp. 100–107, 2018.