Flakify: A Black-Box, Language Model-Based Predictor for Flaky Tests

Sakina Fatima, Taher A. Ghaleb, and Lionel Briand, *Fellow, IEEE*

Abstract—Software testing assures that code changes do not adversely affect existing functionality. However, a test case can be flaky, i.e., passing and failing across executions, even for the same version of the source code. Flaky test cases introduce overhead to software development as they can lead to unnecessary attempts to debug production or testing code. Besides rerunning test cases multiple times, which is time-consuming and computationally expensive, flaky test cases can be predicted using machine learning (ML) models, thus reducing the wasted cost of re-running and debugging these test cases. However, the state-of-the-art ML-based flaky test case predictors rely on pre-defined sets of features that are either project-specific, i.e., inapplicable to other projects, or require access to production code, which is not always available to software test engineers. Moreover, given the non-deterministic behavior of flaky test cases, it can be challenging to determine a complete set of features that could potentially be associated with test flakiness. Therefore, in this article, we propose Flakify, a black-box, language model-based predictor for flaky test cases. Flakify relies exclusively on the source code of test cases, thus not requiring to (a) access to production code (black-box), (b) rerun test cases, (c) pre-define features. To this end, we employed CodeBERT, a pre-trained language model, and fine-tuned it to predict flaky test cases using the source code of test cases. We evaluated Flakify on two publicly available datasets (FlakeFlagger and IDoFT) for flaky test cases and compared our technique with the FlakeFlagger approach, the best state-of-the-art ML-based, white-box predictor for flaky test cases, using two different evaluation procedures: (1) cross-validation and (2) per-project validation, i.e., prediction on new projects. Flakify achieved F1-scores of 79% and 73% on the FlakeFlagger dataset using cross-validation and per-project validation, respectively. Similarly, Flakify achieved F1-scores of 98% and 89% on the IDoFT dataset using the two validation procedures, respectively. Further, Flakify surpassed FlakeFlagger by 10 and 18 percentage points (pp) in terms of precision and recall, respectively, when evaluated on the FlakeFlagger dataset, thus reducing the cost bound to be wasted on unnecessarily debugging test cases and production code by the same percentages (corresponding to reduction rates of 25% and 64%). Flakify also achieved significantly higher prediction results when used to predict test cases on new projects, suggesting better generalizability over FlakeFlagger. Our results further show that a black-box version of FlakeFlagger is not a viable option for predicting flaky test cases.

Index Terms—Flaky tests, software testing, black-box testing, natural language processing, CodeBERT

1 Introduction

Software testing is an essential activity to assure software dependability. When a test case fails, it usually indicates that recent code changes were incorrect. However, it has been observed, in many environments, that test cases can be non-deterministic, passing and failing across executions, even for the same version of the source code. These test cases are referred to as flaky test cases [1], [2], [3]. Flaky test cases can introduce overhead to software development, since they require developers to either (a) debug the production or testing code looking for a bug that might not really exist, or (b) rerun a failed test case multiple times to check if it would eventually pass, thus suggesting that the failure is not due to recent code changes but to the test case itself.

Previous research has investigated the common reasons behind test flakiness, such as concurrency, resource leakage, and test smells. The conventional approach to detect flaky test cases is to rerun them numerous times [4], [5], which is in most practical cases computationally expensive [6] or even impossible. To address this issue, recent studies have proposed approaches using machine learning (ML) models to predict flaky test cases without rerunning them [7], [8], [9], thus proposing a much more scalable and practical solution. Despite significant progress, these approaches (a) rely on production code, which is not always accessible by software test engineers or a scalable solution, or (b) employ project-specific features as flaky test case predictors, which makes them inapplicable to other projects. Moreover, these approaches rely on a limited set of pre-defined features, extracted from the source code of test cases and the system under test. However, when evaluated on realistic datasets, these approaches yield a relatively low accuracy (F1-scores in the range 19%-66%), thus suggesting they may not capture enough information about test flakiness. Finding additional features that could potentially be associated with flaky test cases, preferably based on test code only (black-box), is therefore a research challenge.

- *Sakina Fatima and Taher A. Ghaleb are with the School of EECS, University of Ottawa, Ottawa, ON K1N 6N5, Canada. E-mail: {sfati077, tghaleb}@uottawa.ca.*
- *Lionel Briand is with the School of EECS, University of Ottawa, Ottawa, ON K1N 6N5, Canada, and also with the SnT Centre for Security, Reliability and Trust, University of Luxembourg, 4365 Esch-sur-Alzette, Luxembourg. E-mail: lbriand@uottawa.ca.*

Manuscript received 23 December 2021; revised 19 August 2022; accepted 20 August 2022. Date of publication 24 August 2022; date of current version 18 April 2023.
This work was supported in part by research grant from Huawei Technologies Canada, Mitacs Canada, and in part by the Canada Research Chair and Discovery Grant programs of the Natural Sciences and Engineering Research Council of Canada (NSERC).
(Corresponding author: Taher A. Ghaleb.)
Recommended for acceptance by A. Zaidman.
Digital Object Identifier no. 10.1109/TSE.2022.3201209

0098-5589 © 2022 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission.
See https://www.ieee.org/publications/rights/index.html for more information.
Authorized licensed use limited to: Indian Institute of Information Technology Lucknow. Downloaded on November 20,2024 at 14:05:10 UTC from IEEE Xplore. Restrictions apply.

In this paper, we propose Flakify (Flaky Test Classify), a generic language model-based solution for predicting flaky test cases. Flakify is black-box as it relies exclusively on the source code of test cases (test methods), thus not requiring access to the production code of the system under test. This is important as production code is not always (entirely) accessible to test engineers due, for example, to outsourcing software testing to a third-party. Further, analyzing production code may raise many scalability and practicality issues, especially when applied to large industrial systems using multiple programming languages. In addition, Flakify does not require the definition of features—which are necessarily incomplete—to be used as predictors for flaky test cases. Instead, we used CodeBERT [10], a pre-trained language model, and fine-tuned it to classify test cases as flaky or not based on their source code. To improve Flakify, we further pre-processed test code to remove potentially irrelevant information. We evaluated Flakify on two different datasets: the FlakeFlagger dataset, containing 21,661 test cases collected from 23 Java projects, and the IDoFT dataset, containing 3,862 test cases collected from 312 Java projects. To do this, we used two different evaluation procedures: (1) cross-validation and (2) per-project validation, i.e., prediction on new projects. Our results were compared to FlakeFlagger [7], the best state-of-the-art ML-based predictor for flaky test cases. Specifically, our evaluation addresses the following research questions.

- *RQ1: How accurately can Flakify predict flaky test cases?*
  Flakify achieved promising prediction results when evaluated using two different datasets. In particular, based on cross-validation, Flakify achieved a precision of 70%, a recall of 90%, and an F1-score of 79% on the FlakeFlagger dataset, and a precision of 99%, a recall of 96%, and an F1-score of 98% on the IDoFT dataset. Flakify yielded slightly worse results when predicting flaky tests on new projects, with a precision of 72%, a recall of 85%, and an F1-score of 73% on the FlakeFlagger dataset, and a precision of 91%, a recall of 88%, and an F1-score of 89% on the IDoFT dataset.
- *RQ2: How does Flakify compare to the state-of-the-art predictors for flaky test cases?*
  The best performing model of Flakify achieved a significantly higher precision (70% versus 60%) and recall (90% versus 72%) on the FlakeFlagger dataset in predicting flaky test cases than FlakeFlagger, the best state-of-the-art, white-box approach for predicting flaky test cases. Hence, with Flakify, the cost of debugging test cases and production code is reduced by 10 and 18 percentage points (pp) (a reduction rate of 25% and 64%), respectively, when compared to FlakeFlagger. Moreover, our results show that a black-box version of FlakeFlagger is not a viable option for predicting flaky test cases. Specifically, FlakeFlagger became 39 pp less precise with 20 pp less recall when only black-box features were used as predictors for flaky test cases.
- *RQ3: How does test case pre-processing improve Flakify?*
  Retaining only code statements that are related to a selected set of test smells improved the precision, recall, and F1-score of Flakify by 5 pp and 6 pp on the FlakeFlagger and IDoFT datasets, respectively. The goal was to address a limitation of CodeBERT (and all other language models), which leads to only considering the first 512 tokens in the test source code. This result also confirms the previously reported association of test smells with flaky test cases [7], [9], [11].

Overall, this paper makes the following contributions.

- A generic, black-box, language model-based flaky test case predictor, which does not require rerunning test cases.
- An ML-based classifier that predicts flaky test cases on the basis of test code without requiring the definition of features.
- An Abstract Syntax Tree (AST)-based technique for statically detecting and only retaining statements that match eight test smells in the test code, thus enhancing the application of language models.

The rest of this paper is organized as follows. Section 2 provides background about flaky test cases and language models. Section 3 presents our black-box approach for predicting flaky test cases. Section 4 evaluates our approach, reports experimental results, and discusses the implications of our research.. Section 5 discusses the validity threats to our results. Section 6 reviews and contrasts related work. Finally, Section 7 concludes the paper and suggests future work.

2 Background

In this section, we describe flaky test cases, their root causes, their practical impact, and the strategies to detect them. In addition, we describe pre-trained language models and how they can potentially contribute to predicting flaky test cases.

2.1 Flaky Test Cases

In software testing, a flaky test refers to test cases that intermittently fail and pass across executions, even for the same version of the source code, i.e., non-deterministically behaving test cases [1]. Flaky test cases lead to many problems during software testing, by producing unreliable results and wasting time and computational resources. A flaky test can also fail for different reasons across executions, making it difficult to identify which failures are actually related to faults in the system under test.

Flaky test cases have been reported to be a significant problem in practice at many companies including Google, Huawei, Microsoft, SAP, Spotify, Mozilla, and Facebook [12], [13], [14], [15]. As reported by Google, almost 16% of their 4.2 million test cases are flaky [6]. Microsoft has also reported that 26% of $3.8\ k$ build failures were due to flaky test cases. Many studies have been conducted to study flaky test cases, their causes, and the solutions to address them [1], [2], [4], [7], [8], [9], [11], [16]. Prominent causes of flaky test cases include asynchronous waits, test order dependency, concurrency, resource leakage, and incorrect test inputs or outputs. In addition, flaky test cases were found to be associated with other factors, such as test smells, which are further discussed below.

Authorized licensed use limited to: Indian Institute of Information Technology Lucknow. Downloaded on November 20,2024 at 14:05:10 UTC from IEEE Xplore. Restrictions apply.

TABLE 1
Test Smells Used by FlakeFlagger [7]

Test Smell	Description
Indirect Testing	A test interacts with the class under test using methods from other classes
Eager Testing	A test performs multiple checks for various functionalities
Test Run War	A test allocates files or resources that might be used by other test cases
Conditional Logic	A test uses a conditional *if* statement
Fire and Forget	A test launches background threads or processes
Mystery Guest	A test accesses external resources
Assertion Roulette	A test performs multiple assertions
Resource Optimism	A test accesses external resources without checking their existence

2.2 Flaky Test Case Detection

The most common approach for detecting flaky test cases is by rerunning test cases numerous times to check whether they behave consistently across executions [4], [5]. Though effective, this approach is computationally expensive and not practical in many situations, for example in continuous integration contexts, where builds are submitted automatically and frequently to perform regression testing. To mitigate such cost, other approaches attempted to detect flaky test cases without relying on rerunning them. To that end, characteristics of test cases, such as execution history, coverage information, and static test features, were used to predict whether a test case is flaky or not. Prediction models were built using ML and Natural Language Processing (NLP) techniques [7], [8], [9]. Such techniques require training ML models with pre-defined sets of features used as indicators for test flakiness. Such features commonly present practical limitations, such as (a) their reliance on production code, which is not always accessible or efficiently analyzable by test engineers, and (b) their limited capacity to capture the actual structure or behavior of test cases, such as the use of language keywords [8] or the presence of test smells [7], [9], [11] in test code.

After identifying potentially flaky test cases, developers can focus their investigation on them and, hence, attempt to fix code statements causing such flakiness. Developers may also choose to rerun those specific test cases many more times to verify that they are actually flaky [17]. This is a reasonable undertaking, since test cases predicted as flaky normally represent a small percentage of the entire test suite. This, in turn, significantly eliminates a large part of the effort and time required to investigate or rerun test cases whenever a failure occurs [7].

2.3 Test Smells

Test Smells are inappropriate design or implementation choices made by developers while writing test cases [18]. Though test smells might not harm the functionality of a test case, previous research has reported that they tend to be associated with test flakiness. Test smells were further employed to classify whether a test case is flaky or not. For example, test smells in Table 1 were part of the features used by Alshammari et al. [7] to predict test flakiness. Camara et al. [9] also used a more comprehensive set of test smells for flaky test case prediction. Results showed that *Sleepy Test* and *Assertion Roulette* are among test smells that are highly associated with flaky test cases.

2.4 Pre-Trained Language Models

Much research has been carried out in the field of NLP for developing pre-trained language models. Language models estimate the probability of different linguistic units, i.e., words, symbols, and sequence of them, occurring in a given sentence. There are many language models proposed in the literature, such as BERT [19], ELMo [20], XLNet [21], RoBERTa [22], and VideoBERT [23]. These models were pre-trained, using self-supervised learning, on a large corpus of unlabelled data. For example, BERT was pre-trained using a large dataset of English text collected from books and Wikipedia, whereas VideoBERT was pre-trained using a large dataset of instructional videos collected from YouTube.

Pre-trained language models are often further fine-tuned using a specific, labelled dataset to train neural networks for performing various NLP tasks, such as text classification and entity recognition [24], relation extraction [25], sentence tagging, or next sentence prediction [19]. For example, BERT was fine-tuned to perform sentiment analysis [26], [27], trained on labelled datasets to assign sentiment tags, i.e., positive, negative, or neutral, to a given text. Fine-tuning requires initializing a language model with the same parameters used for pre-training, and then further training the model using labeled data related to a specific task.

Language models usually employ multi-layer transformers as a model architecture to perform many computations in parallel [28]. Transformer models adopt positional embedding to vectorize individual words by considering their positions in a given sequence of words. Thus, unlike Recurrent Neural Networks (RNNS) [29] and Long-Short Term Memory (LSTM) [30], transformer models do not require looking at past hidden states to capture dependencies with previous words in a sequence of words.

Given the wide popularity of language models in various NLP applications, researchers have attempted to apply these language models to programming languages. However, when BERT, for example, was used for detecting the architectural tactics in source code [31], e.g., recognizing software design patterns, the results were relatively worse compared to those obtained when BERT was used for natural language text. To address this issue, recent work proposed pre-training language models on source code written in many programming languages in addition to natural language text [10], [32], [33], [34]. These models are well suited for fine-tuning to perform tasks related to source code. CodeBERT [10] is an example of a language model that was pre-trained on both natural and programming languages.

Authorized licensed use limited to: Indian Institute of Information Technology Lucknow. Downloaded on November 20,2024 at 14:05:10 UTC from IEEE Xplore. Restrictions apply.

2.4.1 CodeBERT

CodeBERT [10] is a language model that was pre-trained on a large, unlabeled dataset containing English text as well as source code written in six different programming languages, namely Java, JavaScript, Python, Ruby, PHP, and Go, obtained from the CodeSearchNet corpus [35]. CodeBERT takes, as input, source code statements and natural language sentences, which are then tokenized using the WordPiece [36] tokenizer. Similar to BERT and RoBERTa, CodeBERT uses a multi-layer bidirectional transformer [28] as model architecture. This transformer is composed of six layers, each of which contains 12 self-attention heads capturing word relationships, a hidden state, and a 768-dimensional vector, as the output of each layer.

CodeBERT also employed Masked Language Modeling (MLM) [19] and Replaced Token Detection (RTD) [37] during pre-training, allowing to take tokens from random positions and masking them with special tokens, which are later used to predict the original tokens. As a result, each token is assigned a vector representation containing information about the token and its position in a given code. The final output of CodeBERT is a single vector representation aggregating all individual vector representations. This vector representation can further be fine-tuned to perform various tasks, e.g., classification. For example, to evaluate the performance of CodeBERT, it was fine-tuned to perform two tasks: (1) code search, i.e., retrieving the most relevant code to a given natural language text; (2) code documentation, i.e., generating a natural language description for a given source code. Moreover, CodeBERT was also adopted to perform classification tasks, such as bug prediction [38] and vulnerability detection [39].

2.4.2 Other Models for Programming Languages

As mentioned above, recently, many language models for programming languages were proposed. For example, GraphCodeBERT [32] was pre-trained on the inherent structure of source code and its data flow showing variables dependencies. Similar to CodeBERT, GraphCodeBERT was used for code search, in addition to code translation and refinement as well as clone detection. Another model for programming languages is TreeBERT [34], which was pre-trained using AST representations of Java and Python source code. TreeBERT was used for code documentation, similar to CodeBERT, in addition to code summarization. There is also CuBERT [33], a programming language model pre-trained using Python source code. CuBERT was used for classification tasks, such as classifying exceptions and variable misuses.

Despite the capabilities of these models, CodeBERT has been the most commonly used language model and we selected it to address our objectives for several reasons presented below.

- The pre-trained CodeBERT model is publicly available.[1]
- Unlike GraphCodeBERT, CodeBERT does not take into consideration the data flow in a given source code, which might not be easy to capture using test code only. For example, unlike local variables, if a global or external variable is used by a test case, GraphCodeBERT cannot identify the type and value of that variable when analyzing test code only.
- Unlike TreeBERT, which requires converting source code into ASTs, CodeBERT only requires source code as input.
- Unlike CuBERT, which was only pre-trained on Python source code without comments, CodeBERT was pre-trained on multiple programming languages using both source code and natural language comments.

3 Black-Box Flaky Test Case Predictor

This section describes our black-box solution for predicting flaky test cases. This is motivated by making such predictions scalable, as white-box analysis of the production source code, especially in the context of large systems, is often not a viable solution.

3.1 CodeBERT for Flaky Test Case Prediction

In this paper, we propose Flakify, a black-box solution for predicting whether a test case is flaky or not. Flakify relies solely on the source code of a test case and does not require to rerun it multiple times. The source code of test cases, i.e., Java test methods, includes the method declaration, body, and it associated Javadoc comments. While several studies have proposed ML techniques to predict flaky test cases, such techniques rely on pre-defined features extracted not only from the source code of test cases but also that of the system under test. However, results [7], [8], [9] suggest those features may not be enough, and finding additional features that could potentially be associated with flaky test cases remains a research challenge given their non-deterministic behavior. Therefore, we employed CodeBERT, the pre-trained language model described above, to perform a binary classification of test cases as *Flaky* or *Non-Flaky*. CodeBERT does not require to define features as it automatically identifies patterns based on the syntax and semantics of a given test code.

CodeBERT starts by converting the source code of a test case into a list of tokens, each of which is converted into an integer vector representation. Finally, an aggregated vector representation is generated as an output of CodeBERT, which is further fine-tuned to classify test cases as *Flaky* or *Non-Flaky*. Fig. 1 presents an example of how the source code of a test case is converted into tokens and then into integer vector representations.

3.1.1 Source Code Tokenization

To transform the source code into tokens, the source code of test cases is tokenized by the WordPiece [36] tokenizer using a pre-generated vocabulary file containing the vocabulary of both English and programming languages used for model pre-training. However, uncommon words, i.e., those that do not exist in the vocabulary file, are separated into several sub-words. For example, the CodeBERT tokenizer splits '`assertThat`' into '`assert`' and '`##that`,' where '`##`' denotes that a token represents a sub-word. Then, if a token is not found in the vocabulary file, the unknown token, `<UNK>`, is used. For each input, two special tokens, `[CLS]`

1. https://huggingface.co/microsoft/CodeBERT-base

Authorized licensed use limited to: Indian Institute of Information Technology Lucknow. Downloaded on November 20,2024 at 14:05:10 UTC from IEEE Xplore. Restrictions apply.

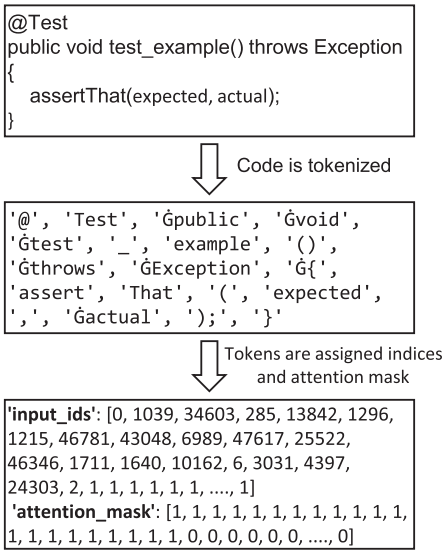

Fig. 1. The process of converting the source code of a test case into a sequence of tokens, where each token is assigned an input index (id) and attention mask. Dots '....' are used to save space, since the actual length is 512. The input id of each token refers to a 768-dimensional vector representation.

and [SEP], are added. Eventually, for a given source code, the tokenizer generates a sequence of tokens in the form of [CLS], $c_1, c_2, .., c_n$, [SEP], where c_i is a code token. The [CLS] token plays an important role in the classification of flaky test cases, as it contains the aggregated vector representation of all the vector representations of the tokens of a given test case. On the basis of that aggregated vector representation, our model classifies a test case as *Flaky* or *Non-Flaky*. [SEP] is just used to mark the end of the sequence of tokens. The tokenizer also adds 'Ġ' in front of each word that is preceded by a whitespace in a statement.

3.1.2 Converting Tokens into Vector Representations

Once the source code tokens are generated, each token, including sub-word, special, and unknown tokens, is mapped to an index, e.g., id 34603 for "Test" in Fig. 1, based on the position and context of each word in a given input. Each token is described by an 768-dimensional integer vector generated during CodeBERT pre-training. Using token padding, the same token length is given to the code of all test cases used as input, e.g., "1" in Fig. 1. However, CodeBERT has a limit of 512 tokens per input. As a result, any token sequence exceeding that limit is truncated, which might lead to removing code statements with potentially relevant information about test flakiness. In addition to *input ids* matching tokens, another list of *attention masks* is generated containing ones and zeros to help the model distinguish between code tokens, which should be given attention, and extra tokens added for padding. Finally, for each test case, token vectors are aggregated to form one vector characterizing the [CLS] token, which is also represented using a 768-sized vector referred to by the first input index '0'.

3.1.3 Fine-Tuning CodeBERT for Flaky Test Classification

CodeBERT was pre-trained with a huge number of parameters, enabling it to recognize the source code structure. As a result, if CodeBERT were to be trained from scratch on our dataset, it would result into over-fitting. To avoid that, CodeBERT, similar to other language models [40], needs to be fine-tuned using data representative of the problem at hand. To do this, we employed CodeBERT as pre-trained and use its outputs, on our dataset, to train a Feedforward Neural Network (FNN) to perform binary classification of test cases as flaky or non-flaky, as shown in Fig. 2.

Fig. 2. Fine-tuning CodeBERT for classifying test cases as *Flaky* or not.

The output of CodeBERT, i.e., the aggregated vector representation of the [CLS] token, is then fed as input to a trained FNN to classify test cases as flaky or not. The FNN contains an *input* layer of 768 neurons, a *hidden* layer of 512 neurons, and an *output* layer with two neurons. We used ReLU [41] as an activation function, which helps to speed up training by transforming the data within layers and output the *input* directly if it is positive or zero otherwise. Then, we added a *dropout* layer [42] to eliminate some neurons randomly from the network, by resetting their weights to zero during the training phase to prevent model over-fitting [43]. We used the *Softmax* function to compute the probability of a test case to be *Flaky* or *Non-Flaky*. We used a learning rate of 10^{-5} using the AdamW optimizer [44] and employed a batch size of two due to computational limitations. Using this configuration, we further trained CodeBERT on our training and validation datasets, which enabled the selection of improved parameter values for weights and biases through back propagation. We then evaluated the model, with the obtained weights, using a test dataset.

3.2 Identifying Test Smells

As indicated above, the 512 token length limit induced by CodeBERT truncates longer test code, which leads to losing potentially relevant information about test flakiness. Therefore, we pre-possessed the source code of test cases to reduce their token length by only retaining information believed to be more relevant to test flakiness. To this end, for test cases exceeding the token length limit, we retained only code statements that match at least one of the eight test smells that were used by FlakeFlagger [7] as predictors for flaky test cases. We also retained the method declaration and the associated Javadoc, since the signature and natural language description, if any, of the test case, might contain key terms or phrases that are likely associated with test flakiness, e.g., *"...failures...unnecessary..."* or *"thread-safe"*.

There exist several open source tools available for detecting test smells [45]. However, these tools, e.g., *tsDetect* [46] and *JNose Test* [47], either rely on production code for detecting test smells or do not detect all test smells that are potentially relevant to test flakiness [7]. While Alshammari et al. [7] detects all the eight test smells shown in Table 1, their technique does so by running test cases and requiring access to the production code for smell detection. Though we were inspired by the heuristics used by Alshammari et al. to detect test smells, given that our approach aims to be black-box, we developed an entirely different technique

Authorized licensed use limited to: Indian Institute of Information Technology Lucknow. Downloaded on November 20,2024 at 14:05:10 UTC from IEEE Xplore. Restrictions apply.

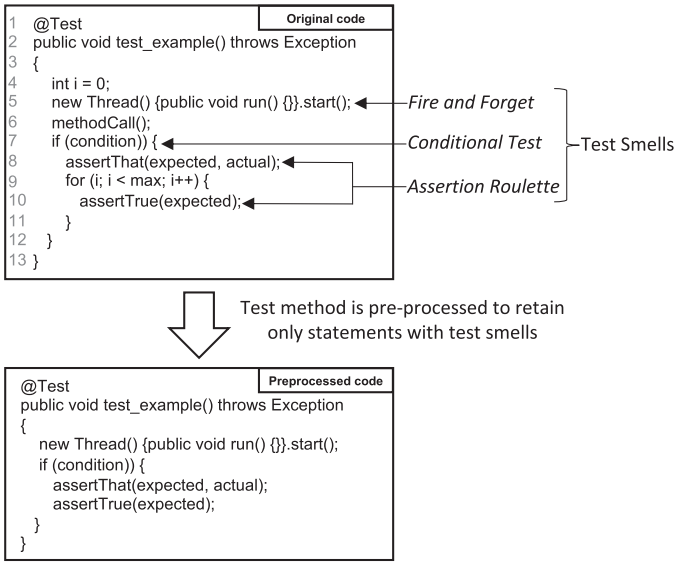

Fig. 3. Example of pre-processing the source code of a test case, which leads to reducing the number of tokens from 62 down to 43.

that detects test smells statically, relying exclusively on test code without requiring to run test cases. Flakify detects all targeted test smells and can be easily extended to detect additional test smells. We used an Abstract Syntax Tree (AST) [48] parser, provided by the Eclipse JDT library,[2] to statically traverse any given test code and retain statements that match any of the targeted test smells. Using this library, each Java file in a test suite is parsed and converted into AST nodes representing different code elements, e.g., method declaration or invocation. Then, an AST visitor is used to traverse those AST nodes. We extended the AST visitor to check the AST nodes related to method declarations and apply heuristics (described below) to detect and retain code statements that match at least one test smell. Such statements are extracted as part of the pre-processed code.

Fig. 3 gives an example of a Java test method, `test_example`, and how it is pre-processed. As we can see, `test_example` has seven different statements, four of them having test smells. In particular, `test_example` contains the following test smells: *Fire and Forget* (line 5 – launching a thread), *Conditional Test* (line 7 – `if` condition), and *Assertion Roulette* (lines 8 and 10 – multiple assertions). As a result, our technique retains only these four statements, which in turn leads to reducing the token length from 62 to 43 (31% reduction rate). We expect our test code pre-processing to help improve the classification performance, since it mitigates the random truncation of code statements induced by CodeBERT.

3.2.1 Heuristics for Detecting Test Smells

To detect test smells in test code, we followed the same detection heuristics as those used by Alshammari et al. [7]. However, different from this work, which extracts test smell information dynamically from the test and production code (code coverage), we detected test smells statically by analyzing the test code only. To this end, we used an Abstract Syntax Tree (AST) [48] parser, provided by the Eclipse JDT library,[3] to traverse any given test code and retain statements that match, according to our heuristics, any of the targeted test smells. Using this library, each Java test file in the test suite is parsed and converted into AST nodes representing different code elements, e.g., method declaration or invocation. While parsing Java test files, not all types are necessarily resolved due to missing production code. We describe below the heuristics used to identify each of the eight test smells presented in Table 1. For each test case, i.e., test method, we analyzed each statement to check whether it matches one of the targeted test smells. If so, we retain that statement as part of the pre-processed test code and otherwise exclude that statement. For some test smells, we added flags, i.e., a Java line comments appended to the end of each statement matching the test smell, to help our fine-tuned model learn about the association of these statements with test flakiness. The test smells used in this work were detected as described below.

- *Indirect Testing:* We check whether a statement invokes a method that belongs to a class other than the test class or the production class under test. Since our approach is black-box, i.e., no access to production code, the production class name is extracted from the test class name by removing the word '`Test`'. This is a commonly used coding convention, but our approach can easily be adapted to other coding conventions in practice [49]. Any statement that is found to invoke such methods is retained and the '`//IT`' flag is added.
- *Eager Testing:* We check whether a test case invokes more than one method belonging to the production class under test as it can introduce confusion to what exactly a test method is testing [45]. If this is the case, we retain the statements invoking these methods, adding the '`//ET`' flag.
- *Test Run War:* We check whether a statement accesses `static` variables that are not declared as *final*, as the value of such variables could be changed by other test cases in different test executions, especially when a test case is order-dependent, which can then cause resource interference during test case execution [7]. Any statement that is found to use one of these variables is retained, adding the '`//RW`' flag.
- *Conditional Logic:* We check whether a statement contains an `if` condition. If so, we retain `if` statements, including their logical expressions. The presence of conditional statements makes test case behavior dependent on their logical expressions, thus making them unpredictable [45]. For the statements inside the `then` and `else` blocks, we only retain those that match one of the eight test smells.
- *Fire and Forget:* We check whether a statement invokes a method that launches a thread by checking if the invoked method belongs to the `java.lang.Thread` class, `java.lang.Runnable` interface, or `java.util.concurrent` package. Thread-related statements make test cases prone to synchronization issues during their execution [11]. If this test smell is present, we retain that statement.
- *Mystery Guest:* We check whether a statement invokes a method that accesses external resources,

2. https://www.eclipse.org/jdt
3. https://www.eclipse.org/jdt

Authorized licensed use limited to: Indian Institute of Information Technology Lucknow. Downloaded on November 20,2024 at 14:05:10 UTC from IEEE Xplore. Restrictions apply.

such as the file system (via `java.io.File`), database system (via `java.sql`, `javax.sql`, or `javax.persistence`), or network (via `java.net` or `javax.net`). Such external resources can introduce stability and performance issues during test case execution [11]. Any statement that is found to use methods that belong to one of these classes or packages is retained.

- *Assertion Roulette:* We check whether a statement performs one of the following assertion mechanisms, including `assertArrayEquals`, `assertEquals`, `assertFalse`, `assertNotNull`, `assertNotSame`, `assertNull`, `assertSame`, `assertThat`, `assertTrue`, and `fail`. If so, the statement is retained. Multiple assert statements in a test method makes it difficult to identify the cause of the failure if just one of the asserts fails [9].
- *Resource Optimism:* We check whether a statement accesses the file system (`java.io.File`) without checking if the path (for either a file or directory) exists. Doing so makes optimistic assumptions about the availability of resources, thus causing non-deterministic behavior of the test case [46]. We check the test initialization method (usually named as `setUp` or containing the `@Before` annotation) for any path checking method, including `getPath()`, `getAbsolutePath()`, or `getCanonicalPath()`. If no such checking is present, the statement is retained, adding the '`//RO`' flag.

4 Validation

This section reports on the experiments we conducted to evaluate how accurate is Flakify in predicting flaky test cases and how it compares to FlakeFlagger as a baseline. We discuss the research questions we address, the datasets used, and the experiment design. Then, we present the results for each research question and discuss their practical implications.

4.1 Research Questions

- *RQ1: How accurately can Flakify predict flaky test cases?*
  The performance of ML-based flaky test predictors can be influenced by the data used for training and the underlying modeling methodology. In this RQ, we evaluate Flakify on two distinct datasets, which differ in terms of numbers of projects, ratios of flaky and non-flaky test cases, and the way flaky test cases were detected. In addition, predicting flaky test cases can be influenced by project-specific information used during model training, which is not available for new projects. Therefore, we evaluate Flakify using two different procedures: 10-fold cross-validation and per-project validation. The former mixes test cases from all projects together to perform model training and testing, whereas the later tests the model on every project such that no information from that project was used as part of model training.
- *RQ2: How does Flakify compare to the state-of-the-art predictors for flaky test cases?*
  Many solutions have been proposed to predict flaky test cases. In this RQ, we compare the performance of our best performing model of Flakify (with test case pre-processing) to two versions (white-box and black-box) of FlakeFlagger, the best flaky test case predictor to date.
  RQ2.1: How accurate is Flakify for flaky test case prediction compared to the best white-box ML-based solution? White-box prediction of flaky test cases requires access to production code, which is not (easily) accessible by software test engineers in many contexts. We assess whether Flakify achieves results that are at least comparable to the best white-box flaky test case predictor. Specifically, we compare the accuracy of the best performing model of Flakify with FlakeFlagger [7], the best white-box solution currently available, on the dataset used by FlakeFlagger. Our motivation is to determine whether black-box solutions, based on CodeBERT, can compete with the state-of-the-art, white-box ones. We compare the results of Flakify and FlakeFlagger on the dataset on which FlakeFlagger was evaluated, hereafter referred to as the FlakeFlagger dataset. We also performed a per-project validation of Flakify compared against FlakeFlagger to assess their relative capability to predict test cases in new projects.
  RQ2.2: How accurate is Flakify for black-box flaky test case prediction compared to the best ML-based solution? Existing black-box flaky test case prediction solutions rely on a limited set of features that are sometimes project-specific or applicable only to a certain programming language, e.g., Java [8], since they were trained on features capturing the keywords of that language. Besides not being generic, the accuracy of these solutions has shown to be very low compared to white-box solutions [7]. Therefore, we compare the accuracy of Flakify with a black-box version of FlakeFlagger, by excluding the features related to production code, such as code coverage features (see Table 2).
- *RQ3: How does test case pre-processing improve Flakify?* The token length limitation of CodeBERT may lead to unintentionally removing relevant information about flaky test cases, which could then impact prediction accuracy. We assess whether the accuracy of Flakify is improved when training the model using pre-processed test cases containing only code statements related to test smells, as opposed to the entire test case code. We fully realize that we may be missing test smells or unintentionally removing relevant statements. But our motivation is to assess the benefits, if any, of our approach to reduce the number of tokens used as input to CodeBERT. We performed this analysis on both the FlakeFlagger and the IDoFT datasets.

4.2 Datasets Collection and Processing

To evaluate Flakify, we used two publicly available datasets for flaky test cases. The first dataset is the FlakeFlagger dataset [7]. The second dataset is the International Dataset of Flaky Tests (IDoFT),[4] which comprises many datasets for

4. https://mir.cs.illinois.edu/flakytests

Authorized licensed use limited to: Indian Institute of Information Technology Lucknow. Downloaded on November 20,2024 at 14:05:10 UTC from IEEE Xplore. Restrictions apply.

TABLE 2
FlakeFlagger Features

Category	Feature	Description
Black-Box	Presence of Test Smells	See Table 1
	Test Lines of Code	Number of lines of code in the body of the test method
	Number of Assertions	Number of assertions checked by the test
	Execution Time	Running time for the test execution
	Libraries	Number of external libraries used by the test
White-Box	Source Covered Classes	Number of production classes covered by each test
	Source Covered Lines	Number of lines covered by the test, counting only production code
	Covered Lines	Number of lines of code covered by the test
	Covered Lines Churn	Churn of covered lines in past 5, 10, 25, 50, 75, 100, 500, and 10,000 commits

flaky test cases used by previous studies on flaky test case prediction [5], [50], [51], [52], [53], [54].

FlakeFlagger Dataset. It is provided by Alshammari et al. [7], containing flakiness information about 22,236 test cases collected from 23 GitHub projects. These projects have different test suite sizes, ranging from 55 to 6,267 (with a median of 430) test cases per project. All projects in the FlakeFlagger dataset are written in Java and use Maven as a build system, and each test case is a Java test method. The dataset contains the source code of each test case and the corresponding features that were computed to train FlakeFlagger. Also, test cases in the dataset were assigned labels indicating whether they are *Flaky* or *Non-Flaky*, which were determined by executing each test case 10,000 times.

When we analyzed the dataset, we identified 453 test cases with missing source code when intersecting test cases in a provided CSV file (called *processed_data*[5]) with those in a provided folder (called *original_tests*[6]) containing their source code. In addition, we identified 122 test cases, in the *original_tests* folder, with empty source code, which we found out were not written in Java.[7] Therefore, we excluded these test cases from our dataset, since they do not add any valuable information regarding our flakiness prediction evaluation. Nine of these test cases were labeled as flaky, three with missing source code and six with empty method body. After excluding test cases with missing and empty code, we obtained 21,661 test cases for our experiments. We compared Flakify and FlakeFlagger using this updated dataset. To pre-process the source code of the test cases (see Section 3.2), we cloned the GitHub repository of each project and extracted the Java classes defining the methods of test cases.

There are 802 test cases in the dataset that are labeled as *Flaky* (with a median of 19 flaky test cases per project), whereas 20,859 test cases are *Non-Flaky*. About 4% of all test cases exceed the 512 limit of CodeBERT when converted into tokens, including 14% of the flaky test cases.

IDoFT Dataset. This dataset contains 3,742 *Flaky* test cases from 314 different Java projects, and collected using different ways, i.e., different runtime environments with different numbers of runs to detect test flakiness. However, we were unable to obtain the test code of 474 test cases (from 2 projects) due to missing GitHub repositories or commits, leaving us with 3,268 *Flaky* test cases from 312 projects. Given that the IDoFT dataset contains no test cases categorized as *Non-Flaky*, we used the fixed versions of 1,263 flaky test cases, from 174 projects, to obtain non-flaky test cases, as recommended by the IDoFT maintainers.[8] To do so, we relied on the provided links to pull requests[9] used for fixing flaky test cases to collect the corresponding code changes. However, of the 1,263 fixed flaky test cases, we found only 594 flaky test cases, from 126 projects, in which the test case code is changed to fix test flakiness. Based on our analysis, the other flaky test cases were fixed in other ways, such as changing the order of test case execution, test configuration, or production code. Such flaky tests are out of the scope of this paper, since we consider only test cases whose test code was fixed, e.g., causes of flakiness related to test smells or other test characteristics. As a result, we added the 594 *Non-Flaky* (fixed) tests to the 3,268 *Flaky* test cases to end up with an updated dataset of 3,862 test cases. Limitations, regarding the causes of flakiness we could not detect, are discussed in Section 5. About 13% of all test cases exceed the 512 limit of CodeBERT when converted into tokens.

We made the updated datasets of FlakeFlagger and IDoFT, including their pre-processed test cases, publicly available in our replication package [55].

4.3 Experiment Design

4.3.1 Baseline

We used the FlakeFlagger approach as a baseline against which we compare the results achieved by Flakify. To this end, we reran the experiments conducted by Alshammari et al. [7] to reproduce the prediction results of FlakeFlagger using their provided replication package.[10] FlakeFlagger was trained and tested using a combination of white-box and black-box features listed in Table 2. These features were selected based on their Information Gain (IG), i.e., only features having an IG ≥ 0.01 were selected for training. Besides reproducing the original results of FlakeFlagger, we also reran the experiments using black-box features only, which was done by excluding all features that required access to

5. https://github.com/AlshammariA/FlakeFlagger/blob/main/flakiness-predicter/result/processed_data.csv
6. https://github.com/AlshammariA/FlakeFlagger/tree/main/flakiness-predicter/input_data/original_tests
7. https://github.com/AlshammariA/FlakeFlagger/pull/4
8. https://github.com/TestingResearchIllinois/IDoFT/issues/566
9. https://mir.cs.illinois.edu/flakytests/fixed.html
10. https://github.com/AlshammariA/FlakeFlagger

Authorized licensed use limited to: Indian Institute of Information Technology Lucknow. Downloaded on November 20,2024 at 14:05:10 UTC from IEEE Xplore. Restrictions apply.

production code. Comparing Flakify with FlakeFlagger is performed on the FlakeFlagger dataset only, as running FlakeFlagger on the IDoFT dataset requires extracting features, both dynamic and static, needed to train FlakeFlagger. To do so, we must access the project's production code and then successfully execute thousands of test cases across hundreds of project versions.

4.3.2 Training and Testing Prediction Models

Training and testing Flakify were conducted using two different procedures, performed independently on the two datasets describe above, as follows.

1st Procedure (Cross-Validation). In this procedure, we evaluated Flakify similarly to how FlakeFlagger was originally assessed. Specifically, we used a 10-fold stratified cross-validation to ensure our model is trained and tested in a valid and unbiased way. For that, we allocated 90% of the test cases for training and 10% for testing our model in each fold. However, different from FlakeFlagger, we employed 20% of the training dataset as a validation dataset, which is required for fine-tuning CodeBERT. Using the validation dataset, we calculated the training and validation loss, which helped obtain optimal weights and stop the training early enough to avoid overfitting.

Given that both of the datasets we used are highly imbalanced—*Flaky* test cases represent only 3.7% of all test cases in the FlakeFlagger dataset and *Non-Flaky* test cases represent only 15% of the IDoFT dataset—we balanced *Flaky* and *Non-Flaky* test cases in the training and validation datasets of FlakeFlagger and IDoFT. Different from FlakeFlagger, which used the synthetic minority oversampling technique (SMOTE) [56], we used random oversampling [57], which adds random copies of the minority class to the dataset. We were unable to use SMOTE, since it requires vector-based features, whereas our model takes the source code of test cases (text) as input [10], [38], as opposed to pre-defined features like FlakeFlagger. Similar to FlakeFlagger, we also performed our experiments using undersampling but this led to lower accuracy. We did not balance the testing dataset to ensure that our model is only tested on the actual set of test cases. This prevents overestimating the accuracy of the model and reflects real-world scenarios where flaky test cases are rarer than non-flaky test cases [7].

2nd Procedure (Per-Project Validation). In this procedure, we evaluated Flakify in a way that yields more realistic results when we predict test cases on a new project, thus evaluating the generalizability of Flakify across projects. To do this, we performed a per-project validation of Flakify on both datasets. In particular, for every project in each dataset, we trained Flakify on the other projects and tested it on that project. This allowed us to evaluate how accurate Flakify is in predicting flaky test cases in one project without including any data from that project during training. We also performed this analysis for FlakeFlagger, on the FlakeFlagger dataset, for the sake of comparison.

4.3.3 Evaluation Metrics

To evaluate the performance of our approach, we used standard evaluation metrics for ML classifiers, including *Precision* (the ability of a classification model to precisely predict flaky test cases), *Recall* (the ability of a model to predict all flaky test cases), and the *F1-Score* (the harmonic mean of precision and recall) [58]. For the per-project validation of Flakify, we computed the overall precision, recall, and F1-score using the prediction results of all projects in the FlakeFlagger and IDoFT datasets. We also computed these metrics individually for those projects that have both *Flaky* and *Non-Flaky* test cases, specifically 23 FlakeFlagger projects and 126 IDoFT projects, along with descriptive statistics, such as mean, median, min, max, 25% and 75% quantiles. We used Fisher's exact test [59] to assess how significant is the difference in proportions of correctly classified test cases between two independent experiments. Note that precision, recall, and F1-score are computed based on such proportions.

In practice, test cases classified as *Flaky* must be addressed by re-running them multiple times or by fixing the root causes of flakiness [6], [12], [60]. Precisely predicting flakiness is therefore important as otherwise time and resources are wasted on re-running and attempting to debug many test cases that are believed to be flaky but are not [16], [61]. According to our industry partner, Huawei Canada, and a Google technical report [6], each flaky test case has to be investigated and re-run by developers. Hence, when we multiply the number of predicted flaky test cases, we proportionally increase the resources associated with re-running and investigating such flaky test cases. Therefore, we assume that the wasted cost of unnecessarily re-running and debugging test cases is inversely proportional to precision

$$Test\ Debugging\ Cost \propto 1 - Precision. \tag{1}$$

On the other hand, it is also important not to miss too many flaky test cases as otherwise time is bound to be wasted on futile attempts to find and fix non-existent bugs in the production code. Thus, we assume that the wasted cost of unnecessarily finding and fixing non-existent bugs in the production code is inversely proportional to recall

$$Code\ Debugging\ Cost \propto 1 - Recall. \tag{2}$$

We acknowledge that the above metrics are surrogate measures for cost and that there are significant differences between individual flaky tests; however, they are reasonable and useful approximations on large test suites for the purpose of comparing classification techniques. We used FlakeFlagger as baseline to compute the reduction rate of test and code debugging costs, by dividing the difference in cost between Flakify and FlakeFlagger by the cost of FlakeFlagger.

4.4 Results

4.4.1 RQ1 Results

Table 3 shows the prediction results (in terms of precision, recall, and F1-score) of Flakify using both the full and preprocessed test code from the FlakeFlagger and IDoFT datasets, based on cross-validation. Overall, Flakify achieved promising prediction results using both datasets, with a precision of 70%, a recall of 90%, and an F1-score of 79% on the FlakeFlagger dataset, and a precision of 99%, a recall of 96%, and an F1-score of 98% on the IDoFT dataset. The higher results achieved by Flakify on the IDoFT dataset over those achieved on the FlakeFlagger dataset is probably

Authorized licensed use limited to: Indian Institute of Information Technology Lucknow. Downloaded on November 20,2024 at 14:05:10 UTC from IEEE Xplore. Restrictions apply.

TABLE 3
Results of Flakify (using Full Code and Pre-Processed) Compared to FlakeFlagger (White-Box and Black-Box Versions)

Approach	Dataset	Model	Precision	Recall	F1-Score
Flakify	FlakeFlagger dataset	Full code	65%	85%	74%
		Pre-processed code	**70%**	**90%**	**79%**
	IDoFT dataset	Full code	98%	95%	92%
		Pre-processed code	**99%**	**96%**	**98%**
FlakeFlagger	FlakeFlagger dataset	White-box version	60%	72%	65%
		Black-box version	21%	52%	30%

TABLE 4
Summary of the Per-Project Prediction Results of Flakify on the FlakeFlagger and IDoFT Datasets

Dataset	Metric	Min	25%	Mean	Median	75%	Max
FlakeFlagger dataset	Precision	6%	58%	**72%**	79%	91%	100%
	Recall	1%	87%	**85%**	95%	100%	100%
	F1-Score	2%	63%	**73%**	83%	94%	100%
IDoFT dataset	Precision	66%	100%	**91%**	100%	100%	100%
	Recall	14%	94%	**88%**	100%	100%	100%
	F1-Score	25%	95%	**89%**	100%	100%	100%

due to the fact that the IDoFT dataset contains many more flaky test cases than FlakeFlagger, which helped during model training. Moreover, the non-flaky test cases in the IDoFT dataset were labeled based on developer's fixes addressing the causes of flakiness in the test code, unlike the non-flaky test cases in the FlakeFlagger dataset whose labels were based on 10,000 runs performed by Alshammari et al. [7], which may not have been enough to fully expose test flakiness. This also helped during model training of Flakify.

Table 4 reports the per-project prediction results of Flakify on the FlakeFlagger dataset. Overall, as expected, Flakify achieved slightly lower precision (72%), recall (85%), and F1-score (73%) than the cross-validation results on the FlakeFlagger dataset. Similarly, Flakify achieved slightly worse precision (91%), recall (88%), and F1-score (89%) on the IDoFT dataset. Table 5 shows descriptive statistics for the per-project prediction results of Flakify for individual projects of the FlakeFlagger dataset (due to space limitations, we provide individual per-project prediction results of Flakify on the IDoFT dataset in our replication package [55]). Our analysis of individual per-project prediction results revealed a high performance of Flakify on the majority of projects. This result suggests that Flakify helps build models that are generalizable across projects, thus making it applicable to new projects where no historical information about test flakiness exists. In short, Flakify is capable to learn about test flakiness through data collected from other projects to predict flaky test cases in new projects.

4.4.2 RQ2 Results

Table 3 presents the prediction results of Flakify, using both full code and pre-processed test code, and FlakeFlagger, using both white-box and black-box versions, for the FlakeFlagger dataset.

RQ2.1 results. For FlakeFlagger, we obtained results close to those reported in the original study, with a slight decrease in F1-score (1%), which is likely due to removing test cases with missing test code. Flakify achieved much better results with a precision of 70% (+10 pp), a recall of 90% (+18 pp), and an F1-score of 79% (+14 pp). These results clearly show that Flakify, though being black-box and relying exclusively on test code, significantly surpasses FlakeFlagger in accurately predicting flaky test cases. Statistically, the proportion of correctly predicted test cases using Flakify is significantly higher than that obtained with FlakeFlagger (Fisher-exact p-value < 0.0001).

The number of true positives obtained by FlakeFlagger was 574, whereas Flakify increased that number to 721. This indicates that Flakify can potentially reduce the test debugging cost by 10 pp, as defined above, when compared to FlakeFlagger (a reduction rate of 25%). Similarly, Flakify reduces the number of false negatives to 81 from 227 with FlakeFlagger, thus decreasing the code debugging cost by 18 pp, as defined above (a reduction rate of 64%).

Table 5 shows the comparison of per-project prediction results between Flakify and FlakeFlagger. Overall, Flakify achieves a high accuracy, with a precision of 72% (+57 pp)), a recall of 85% (+71 pp)), and an F1-score of 73% (+66 pp)), which, once again, significantly outperforms FlakeFlagger. Looking at the individual prediction results of the projects, we observe that the accuracy of Flakify is largely consistent across projects, with a few exceptions, whereas FlakeFlagger performed poorly on the majority of projects. Further, Flakify performs better than FlakeFlagger for almost all projects except two: `incubator-dubbo` and `spring-boot` where both techniques fare poorly.

To understand the reasons behind such degraded performance for these two projects, we performed a hierarchical clustering of the 23 projects. We used different metrics that capture the characteristics of each project, such as the

Authorized licensed use limited to: Indian Institute of Information Technology Lucknow. Downloaded on November 20,2024 at 14:05:10 UTC from IEEE Xplore. Restrictions apply.

TABLE 5
Results of the Per-Project Prediction for Flakify and FlakeFlagger on the FlakeFlagger Dataset

Project	Precision		Recall		F1-Score	
	Flakify	FlakeFlagger	Flakify	FlakeFlagger	Flakify	FlakeFlagger
achilles	100%	0%	100%	0%	100%	0%
activiti	80%	2%	90%	94%	85%	4%
alluxio	99%	100%	100%	13%	99%	24%
ambari	75%	39%	95%	61%	84%	47%
assertj-core	25%	0%	100%	0%	40%	0%
commons-exec	25%	0%	100%	0%	40%	0%
elastic-job-lite	50%	0%	100%	0%	60%	0%
handlebars.java	30%	0%	100%	0%	50%	0%
hbase	79%	72%	98%	33%	88%	45%
hector	100%	0%	93%	0%	96%	0%
http-request	88%	0%	88%	0%	88%	0%
httpcore	74%	7%	90%	4%	81%	5%
incubator-dubbo	6%	7%	16%	32%	9%	12%
java-websocket	95%	0%	95%	0%	95%	0%
logback	85%	0%	81%	0%	83%	0%
ninja	100%	0%	100%	0%	100%	0%
okhttp	78%	100%	85%	2%	81%	4%
orbit	88%	0%	100%	0%	93%	0%
spring-boot	40%	9%	1%	3%	2%	4%
undertow	75%	7%	85%	43%	79%	12%
wildfly	65%	6%	91%	26%	76%	10%
wro4j	88%	1%	100%	19%	94%	3%
zxing	100%	0%	50%	0%	66%	0%
Overall	**72%**	15%	**85%**	14%	**73%**	7%

For every project, we trained models on all other projects and tested them on that project.

number of test cases, number of flaky test cases, and frequency of test smells in each project. However, our clustering results were inconclusive, thus revealing no significant differences between the two projects and the other projects. As reported by Alshammari et al. [7], each project can have distinct characteristics, e.g., environmental setup and testing paradigm, that make it difficult to develop a general-purpose flaky test case predictor. For example, the `spring-boot` project has the highest number of flaky test cases among all projects, representing 20% of all flaky test cases in the dataset. This, in turn, can influence model training when the model was tested for `spring-boot`. In addition, the variation in prediction results can be a result of a possible mislabeling of test cases as *Flaky* and *Non-Flaky* in some projects, since some test cases may still exhibit flakiness behavior if executed more than 10,000 executions, for example. Finally, test flakiness can also occur due to the use of network APIs or dependency conflicts [17], which were not taken into account when predicting flaky test cases.

RQ2.2 results. As shown in Table 3, we observe a considerable decline in the accuracy for the black-box version of FlakeFlagger when compared to its original, white-box version, i.e., 39 pp less precise with a 54 pp decrease in F1-score. Specifically, black-box FlakeFlagger correctly predicted a significantly lower proportion of test cases than both Flakify and the original, white-box version of FlakeFlagger (Fisher-exact p-values < 0.0001). As a possible explanation, based on the results of FlakeFlagger regarding the importance of features in predicting flaky test cases [7], the majority of features having high IG values were based on source code coverage. Hence, removing those features, to make FlakeFlagger black-box, is expected to significantly decrease its prediction power. The difference in accuracy between Flakify and the black-box version of FlakeFlagger is rather striking, with a large improvement of +49% in F1-score (Fisher-exact p-value < 0.0001). FlakeFlagger is therefore not a viable black-box option to predict flaky test cases.

4.4.3 RQ3 Results

With no code pre-processing, 898 (4%) of the test cases of the FlakeFlagger dataset and 505 (13%) of the test cases of the IDoFT dataset were truncated by CodeBERT to generate tokens of size 512. Such arbitrary code truncation is likely to affect how accurately Flakify can predict flaky test cases. Pre-processing test cases (see Section 3.2) led to reducing the number of test cases being truncated to only 40 (from 898) in the FlakeFlagger dataset and 87 (from 505) in the IDoFT dataset, a large difference. As a result, we observe in Table 3 that, with pre-processed test cases, Flakify predicted flaky test cases with 5 pp higher F1-score on the FlakeFlagger dataset and 6 pp higher F1-score on the IDoFT dataset. This corresponds to a significantly higher proportion of correctly predicted test cases (Fisher-exact p-value $= 0.0008$) for the FlakeFlagger dataset. In practice, the impact of pre-processing is expected to vary depending on the token length distribution of test cases. This result suggests that retaining statements related to test smells in the test code contributed to making Flakify more accurate, which also confirms the association of test smells with flaky test cases reported by prior research [9].

4.5 Discussion

More Accurate Predictions With Easily Accessible Information. Our results showed that our black-box prediction of flaky

Authorized licensed use limited to: Indian Institute of Information Technology Lucknow. Downloaded on November 20,2024 at 14:05:10 UTC from IEEE Xplore. Restrictions apply.

test cases performs significantly better than a white-box, state-of-the-art approach. This not only enables test engineers to predict flaky test cases without rerunning test cases, but also without accessing the production code of the system under test, a significant practical advantage in many contexts. The highest accuracy of our Flakify was achieved by only retaining relevant code statements matching eight test smells. Yet, there is still room for improvement in terms of accuracy, which could be achieved by retaining more relevant statements based on additional test smells. For example, retaining code statements related to other common flakiness causes [16], such as concurrency and randomness, could further improve flaky test case predictions. However, the more code statements we retain, the more tokens to be considered by CodeBERT, which might lead to many test cases exceeding their token length limit, thus truncating other useful information. Hence, retaining additional code statements is a trade-off and should carefully be performed in balance with the resulting token length of test cases. Moreover, building a white-box flaky test predictor, by considering both production and test code, is not always technically feasible, since the production code is not always available to test engineers and, when possible, code coverage can be expensive and not scalable on large systems, especially in a continuous integration context. Considering the production code also makes it impractical to build language model-based predictors for flaky test cases, given the token length limitation of language models in general, and CodeBERT in particular. Nevertheless, future research should assess the practicability of white-box, model-based flaky test prediction, and should investigate further code pre-processing methods to make the use of language models more applicable in practice.

Practical Implications of Imperfect Prediction Results. Though Flakify surpassed the best state-of-the-art solution in predicting flaky test cases, both in terms of precision and recall, a precision of 70% is still not satisfactory, since misclassifying non-flaky test cases as flaky leads to additional, unnecessary cost, e.g., attempting to fix the test cases incorrectly predicted as flaky. Also, with a recall of 90%, we miss 10% of flaky test cases, leading to wasted debugging cost. If we assume that precision should be prioritized over recall, we can increase the former by restricting flaky test case predictions to those test cases with highest prediction confidence, at the expense of a lower recall. For example, this can be achieved by adjusting the classification threshold for flaky test cases to 0.60 or 0.70, instead of the default threshold of 0.50. Nevertheless, given that the predicted probabilities generated by the neural network in Flakify are over confident due to the use of the *Softmax* function in the last layer [62], i.e., probabilities are either close to 0.0 or 1.0, we were unable to perform such analysis. Therefore, future research should employ techniques for calibrating the predicted probabilities [63] and enable threshold adjustments when classifying flaky test cases.

Deployment of a Flaky Test Case Predictor in Practice. Flakify can be deployed in Continuous Integration (CI) environments to help detect flaky test cases. One could argue that the CI build history can be used as reference to conclude whether a test case is flaky or not. However, regular test case executions across builds may not entirely solve the problem, since differences in test case verdicts, i.e., pass or fail, can be due to differences in builds rather than flakiness. Therefore, test engineers can use the prediction results obtained from Flakify to fix test cases that are predicted as flaky, e.g., by eliminating the presence of test smells, or otherwise rerun them a larger number of times, using the same code version, to verify whether a test case is actually flaky or not. More specifically, Flakify helps test engineers focus their attention on a small subset of test cases that are most likely to be flaky in a CI build. As our results show, Flakify significantly reduces the cost of debugging test and production code, both in terms of human effort and execution time. This makes Flakify an important strategy in practice to achieve scalability, especially when applied to large test suites. Moreover, the test smell detection capability of Flakify helps to inform test engineers about possible causes of flakiness that need to be addressed.

5 Threats to Validity

This section discusses the potential threats to the validity of our reported results.

5.1 Construct Validity

Construct threats to validity are concerned with the degree to which our analyses measure what we claim to analyze. In our study, to pre-process test cases, we used heuristics to retain code statements that match at least one of the eight test smells shown in Table 1. However, our heuristics might have missed some code statements having test smells and this could have led to suboptimal results when applying our approach. To mitigate this issue, though our approach to identify test smells is entirely different, we relied on the same heuristics as those used by Alshammari et al. [7]. These heuristics assume commonly used coding conventions that might not be followed in all test suites. For example, we assumed that the test class name contains the production class name with the word '*Test*'. However, such heuristics can easily be adapted to other coding conventions in practice. We also manually checked a random sample of test cases to verify that pre-processed code contains, as expected, only test smells-related code statements and does not dismiss any of them. We have made the tool we developed to detect test smells publicly available in our replication package [55].

5.2 Internal Validity

Internal threats to validity are concerned with the ability to draw conclusions from our experimental results. In our study, we used CodeBERT to perform a binary classification of test cases as *Flaky* or *Non-Flaky*. However, due to the token length limit of CodeBERT, the source code of some test cases was truncated, possibly leading to discarding relevant information about test flakiness. To mitigate this issue, we pre-processed the source code of test cases to retain only code statements related to test smells. Doing so did not only reduce the token length of test cases, but also improved the prediction power of our approach. However, our pre-processing may not be perfect or complete as it can lead to losing other relevant information. Future research should investigate whether retaining additionally relevant information to

Authorized licensed use limited to: Indian Institute of Information Technology Lucknow. Downloaded on November 20,2024 at 14:05:10 UTC from IEEE Xplore. Restrictions apply.

flaky test cases leads to improving prediction results, e.g., statements related to common flakiness causes, such as synchronous or platform-dependent operations.

Moreover, our prediction results were compared with those of FlakeFlagger. But FlakeFlagger used white-box features, whereas our approach is black-box and the comparison may not be entirely meaningful. To mitigate this issue, we also compared our results with a black-box version of FlakeFlagger in which we removed any features requiring access to production code. In both cases, our approach obtained significantly higher prediction results than FlakeFlagger. We did not compare our results with other black-box approaches, e.g., vocabulary-based [8], since they are project-specific and did not achieve good results on the FlakeFlagger dataset [7].

Finally, in our analysis, the cost of debugging the production or testing code assumes that test engineers address all test cases predicted as flaky. However, test engineers may choose to ignore a flaky test case, either by removing or skipping it, thus not introducing any cost. Yet, we believe that every flaky test case should be carefully addressed by test engineers, since ignoring test cases can lead to other kinds of costs, such as overlooked system faults.

5.3 External Validity

External threats are concerned with the ability to generalize our results. Our study is based on data collected by Alshammari et al. [7], which was obtained by rerunning test cases 10,000 times. Such data is of course not perfect as some test cases that were not found to be flaky could have been if rerun more times. To mitigate this threat, we used the same dataset for comparing Flakify with the baseline approach, FlakeFlagger. We also filtered out test cases which, to our surprise, had no source code in the dataset. Further, the FlakeFlagger and IDoFT datasets contain test cases from projects that are exclusively written in Java, which might affect the generalizability of our results. To mitigate this issue, we used CodeBERT, which was trained on six programming languages. Hence, we believe our approach would be applicable to projects written in other programming languages as well, given an appropriate tool to identify test smells.

Moreover, CodeBERT was pre-trained on production source code only, i.e., source code related to test suites was not part of pre-training, making it unable to recognize test-specific structure and vocabulary, e.g., assertions. This can potentially increase token length, since test-specific key terms are decomposed into multiple tokens instead of one. For example, CodeBERT converts `assertEquals` into three tokens: `assert`, `##equal`, and `##s`, rather than just one token. Our pre-processing of the source code of test cases helped to mitigate the issue of token length; yet, future work should aim at pre-training CodeBERT on test code in addition to production code.

Finally, the IDoFT dataset has shown that a significant number of test cases are flaky due to reasons unrelated to the test code. In situations where this is common, this is obviously a limitation of any black-box approach like Flakify relying exclusively on test code. In our evaluation, we did not consider such flaky test cases, but rather those whose causes of flakiness were in the test code, which were confirmed and manually fixed by developers, and thus considered in this paper as non-flaky. This helped during model training of Flakify on this dataset, which resulted in a higher prediction accuracy than those on the FlakeFlagger dataset.

6 Related Work

Flaky test detection has been an active area of research where many techniques were proposed to detect flaky test cases [16]. Overall, these techniques can be classified into two groups: dynamic techniques, which require executing test cases to determine whether they are flaky or not, and static techniques, which rely only on the source code of test cases or the system under test. In this section, we review the flaky test detection techniques while comparing and contrasting them to our approach.

6.1 ML-Based Flaky Test Case Prediction

A common approach to detect flaky test cases is to re-run test cases multiple times [1], [16], which is computationally expensive. To address this issue, recent research has proposed the use of ML techniques for predicting flaky test cases, enabling test engineers to re-run only those test cases that are predicted to be flaky, thus reducing the cost of unnecessary debugging of test cases or production code.

Alshammari et al. [7] proposed an innovative approach to predict flaky test cases using dynamically computed features capturing code coverage, execution history, and test smells. They re-ran test cases 10,000 times to identify whether a test case was flaky or not and thus establish a ground truth. Their prediction model predicted flaky test cases with an F1-score of 0.65, leaving significant room for improvement. However, some of the significant features required access to production files which, as discussed above, are not always accessible by test engineers or may not be computable in a scalable way in many practical contexts. Further, when only black-box features (see Table 2) were used, the F1-score decreased by 35 pp. In contrast, our approach achieved more accurate prediction results, with an F1-score of 0.79, while using test code only, thus offering a favorable black-box alternative.

In addition, Pontillo et al. [11] proposed an approach to identify the most important factors associated with flaky test cases using the iDFlakies dataset [5]. They used logistic regression to model flaky test cases using features that were statically computed using production code, e.g., code coverage, and test code, e.g., test smells. They found that code complexity (both production and test code), assertions, and test smells are associated with test flakiness.

Another approach was proposed by Pinto et al. [8] in which Java keywords were extracted from test code and employed as vocabulary features to predict test flakiness. Further, their study relied on the dataset of DeFlaker [4], in which test cases were re-run less than 100 times to establish the ground truth. Despite high accuracy results (F1-score = 0.95) on their dataset, their approach achieved much worse results (F1-score = 0.19) when using the dataset provided by Alshammari et al. [7]. In addition, their models were language- and project-specific, since most of the significant features for predicting flaky test cases were related to Java keywords, e.g., *throws*, or specific variable names, e.g., *id*. In

Authorized licensed use limited to: Indian Institute of Information Technology Lucknow. Downloaded on November 20,2024 at 14:05:10 UTC from IEEE Xplore. Restrictions apply.

contrast, while our approach relies exclusively on test code, it builds a generic model to predict flakiness, based on features that are neither language- nor project-dependent, and achieved much better prediction results when using the FlakeFlagger dataset used by Alshammari et al. [7].

Moreover, Haben et al. [15] and Camara et al. [64] replicated the study by Pinto et al. using other datasets containing projects written in other programming languages, e.g., Python. They found that vocabulary-based approaches are not generalizable, especially when performing inter-project flaky test case predictions, since new vocabulary is needed for any new project or programming language. Haben et al. also showed that combining the vocabulary-based features with code coverage features does not significantly improve the prediction accuracy of such an approach.

In summary, unlike the ML-based approaches above, our approach is generic, black-box, and language model-based, thus not requiring access to production code or pre-definition of features. Instead, our approach relies solely on test code to predict whether a test case is flaky or not.

6.2 Flaky Test Case Prediction Using Test Smells

Camara et al. [9] proposed an approach for predicting test flakiness using test smells as prediction features. These features require access to the production code and can be extracted using tsDetect [46], a tool for detecting test smells, that was applied to the DeFlaker dataset [4]. Their study yielded a relatively high prediction accuracy (F1-score = 0.83). Alshammari et al. [7] also relied on test smells as part of their features for predicting flaky test cases. However, the information gain of test smell features tended to be much lower than code coverage features, suggesting they are less significant flaky test case predictors. In Flakify, we also relied on the test smells used by Alshammari et al. [7]. However, they were not used as features but to exclusively retain relevant test code statements for fine-tuning our CodeBERT model. Doing so improved the accuracy of Flakify, thus reducing the cost of rerunning or debugging test cases.

6.3 Flaky Test Detection at Run Time

Memon et al. [65] used a simple dynamic pattern matching approach to detect flaky test cases at Google by simply searching for certain textual patterns in test execution logs, e.g., *pass-fail-pass*, to identify whether a test case is flaky or not. The accuracy of detecting flaky test cases using this approach was 90%. Similarly, Kowalczyk et al. [66] detected flaky test cases at Apple by analyzing the behavior of test cases using two scores: *Flip rate*, which measures the rate at which a test case alternates between *pass* and *fail*, and *Entropy*, which quantifies the uncertainty of a test case. An aggregated value of these two scores was used to generate flakiness ranks for test cases, which were then used to represent test flakiness, distributed across the test cases in different services at Apple. This technique marked 44% of test failures as flaky with less than 1% loss in fault detection. The above approaches require test cases to be executed many times to determine whether they are flaky, which is often not practical for large industrial projects. Unlike these approaches, Flakify is able to predict flaky test cases without executing them, relying exclusively on test code.

Bell et al. [4] proposed DeFlaker, a tool for detecting flaky test cases using coverage information about code changes. In particular, a test case is labeled as flaky if it fails and does not cover any changed code. Out of 4,846 test failures, DeFlaker was able to label 39 pp of them as flaky, with a 95.5% recall and a false positive rate of 1.5%, outperforming the default way of detecting flaky test cases, i.e., by rerunning test cases using Maven [67]. Different from DeFlaker, Lam et al. [5] proposed iDFlakies, which detects test flakiness by re-running test cases in random orders. This framework was used to construct a dataset containing 422 flaky test cases, with almost half of them being order-dependent.

The above approaches either depend on rerunning test cases multiple times, execution history (not available for new test cases), or production code, e.g., coverage information. In contrast, Flakify does not require repeated executions of test cases or any information about the production code, including code coverage.

7 Conclusion

In this paper, we proposed Flakify, a black-box solution for predicting flaky test cases using only the source code of test cases, as opposed to the system under test. Further, it does not require to rerun test cases multiple times and does not entail the definition of features for ML prediction.

We used CodeBERT, a pre-trained language model, and fine-tuned it to classify test cases as flaky or not based exclusively on test source code. We evaluated our work on two distinct datasets, namely the FlakeFlagger and IDoFT datasets, using two different evaluation procedures: (1) cross-validation and (2) per-project validation, i.e., prediction on new projects. In addition, we pre-processed this source code by retaining only code statements that match eight test smells, which are expected to be associated with test flakiness. This aimed at addressing a limitation of CodeBERT (and related language models), which can only process 512 tokens per test case. We evaluated our approach in comparison with both white-box and black-box versions of FlakeFlagger, the best state-of-the-art, ML-based flaky test case predictor. The main results of our study are summarized as follows:

- Flakify achieves promising results on two different datasets (FlakeFlagger and IDoFT) and under two different evaluation procedures, one assuming Flakify predicts test cases from a new project and the other one simply relying on cross-validation.
- When predicting test cases in new projects, the accuracy of Flakify is slightly lower but still close to cross-validation results.
- With cross-validation, Flakify reduces by 10 pp and 18 pp of the cost bound to be wasted by the original, white-box version of FlakeFlagger due to unnecessarily debugging test cases and production code, respectively.
- Similar to cross-validation results, Flakify also significantly outperforms FlakeFlagger when predicting flaky test cases in new projects, for which the model was not trained.

Authorized licensed use limited to: Indian Institute of Information Technology Lucknow. Downloaded on November 20,2024 at 14:05:10 UTC from IEEE Xplore. Restrictions apply.

- A black-box version of FlakeFlagger is not a viable option to predict flaky test cases as it is too inaccurate.
- When retaining only code statements related to test smells, Flakify predicted flaky test cases with 5 pp and 6 pp higher F1-score on the FlakeFlagger and IDoFT datasets, respectively.

Overall, existing public datasets [4], [5], [7], [15] are not fully adequate to appropriately evaluate flaky test case prediction approaches, since the ratio of flaky test cases tends to be very low. In addition, flaky test cases in these datasets were detected by rerunning test cases numerous times while monitoring their behavior across executions, a technique that may be inaccurate. Further, many open source projects nowadays adopt Continuous Integration (CI), which provides extensive test execution histories. Given the frequency of test executions in CI and the high workload on CI servers, test cases might expose further flakiness behaviors due to causes that may not be revealed when running test cases on machines dedicated to test execution [68], [69]. Therefore, we plan in the future to build a larger dataset of flaky test cases in a CI context.

Last, a significant proportion of flaky tests can be due to problems in the production code and cannot be addressed by black-box models. Therefore, in the future, we need to devise light-weight and scalable approaches to address such causes of flakiness.

Acknowledgments

The experiments conducted in this work were enabled in part by WestGrid (https://www.westgrid.ca) and Compute Canada (https://www.computecanada.ca). Moreover, we are grateful to the authors of FlakeFlagger and the maintainers of the IDoFT dataset, who have responded to our multiple inquiries for clarifications about the datasets.

References

[1] B. Zolfaghari, R. M. Parizi, G. Srivastava, and Y. Hailemariam, "Root causing, detecting, and fixing flaky tests: State of the art and future roadmap," *Softw.: Pract. Exp.*, vol. 51, no. 5, pp. 851–867, 2021.

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

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

[4] J. Bell, O. Legunsen, M. Hilton, L. Eloussi, T. Yung, and D. Marinov, "DeFlaker: Automatically detecting flaky tests," in *Proc. IEEE/ACM 40th Int. Conf. Softw. Eng.*, 2018, pp. 433–444.

[5] W. Lam, R. Oei, A. Shi, D. Marinov, and T. Xie, "iDFlakies: A framework for detecting and partially classifying flaky tests," in *Proc. 12th IEEE Conf. Softw. Testing, Validation Verification*, 2019, pp. 312–322.

[6] J. Micco, "Advances in continuous integration testing at Google," 2018. [Online]. Available: https://research.google/pubs/pub46593

[7] A. Alshammari, C. Morris, M. Hilton, and J. Bell, "FlakeFlagger: Predicting flakiness without rerunning tests," in *Proc. IEEE/ACM 43rd Int. Conf. Softw. Eng.*, 2021, pp. 1572–1584.

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

[9] B. Camara, M. Silva, A. Endo, and S. Vergilio, "On the use of test smells for prediction of flaky tests," in *Proc. Braz. Symp. Systematic Autom. Softw. Testing*, 2021, pp. 46–54.

[10] Z. Feng et al., "CodeBERT: A pre-trained model for programming and natural languages," in *Proc. Findings Assoc. Comput. Linguistics: Empir. Methods Natural Lang. Process.*, 2020, pp. 1536–1547.

[11] V. Pontillo, F. Palomba, and F. Ferrucci, "Toward static test flakiness prediction: A feasibility study," in *Proc. 5th Int. Workshop Mach. Learn. Techn. Softw. Qual. Evol.*, 2021, pp. 19–24.

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

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

[14] T. Bach, A. Andrzejak, and R. Pannemans, "Coverage-based reduction of test execution time: Lessons from a very large industrial project," in *Proc. IEEE Int. Conf. Softw. Testing, Verification Validation Workshops*, 2017, pp. 3–12.

[15] G. Haben, S. Habchi, M. Papadakis, M. Cordy, and Y. L. Traon, "A replication study on the usability of code vocabulary in predicting flaky tests," in *Proc. IEEE/ACM 18th Int. Conf. Mining Softw. Repositories*, 2021, pp. 219–229.

[16] O. Parry, G. M. Kapfhammer, M. Hilton, and P. McMinn, "A survey of flaky tests," *ACM Trans. Softw. Eng. Methodol.*, vol. 31, no. 1, pp. 1–74, 2021.

[17] O. Parry, G. M. Kapfhammer, M. Hilton, and P. McMinn, "Surveying the developer experience of flaky tests," in *Proc. IEEE/ACM Int. Conf. Softw. Eng.: Softw. Eng. Pract.*, 2022, pp. 253–262.

[18] A. Van Deursen, L. Moonen, A. Van Den Bergh, and G. Kok, "Refactoring test code," in *Proc. 2nd Int. Conf. Extreme Program. Flexible Processes Softw. Eng.*, 2001, pp. 92–95.

[19] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of deep bidirectional transformers for language understanding," in *Proc. Conf. North Amer. Chapter Assoc. Comput. Linguistics: Human Lang. Technol.*, 2019, pp. 4171–4186.

[20] M. E. Peters et al., "Deep contextualized word representations," in *Proc. Conf. North Amer. Chapter Assoc. Comput. Linguistics: Human Lang. Technol.*, 2018, pp. 2227–2237.

[21] Z. Yang et al., "XLNet: Generalized autoregressive pretraining for language understanding," in *Proc. Adv. Neural Inf. Process. Syst.*, 2019, pp. 5753–5763.

[22] Y. Liu et al., "RoBERTa: A robustly optimized BERT pretraining approach," 2019, *arXiv:1907.11692*.

[23] C. Sun, A. Myers, C. Vondrick, K. Murphy, and C. Schmid, "VideoBERT: A joint model for video and language representation learning," in *Proc. IEEE/CVF Int. Conf. Comput. Vis.*, 2019, pp. 7464–7473.

[24] D. Nadeau and S. Sekine, "A survey of named entity recognition and classification," *Lingvisticae Investigationes*, vol. 30, no. 1, pp. 3–26, 2007.

[25] N. Bach and S. Badaskar, "A review of relation extraction," *Literature Rev. Lang. Statist.*, vol. II, no. 2, pp. 1–15, 2007.

[26] H. Xu, B. Liu, L. Shu, and P. S. Yu, "BERT post-training for review reading comprehension and aspect-based sentiment analysis," in *Proc. Conf. North Amer. Chapter Assoc. Comput. Linguistics: Human Lang. Technol.*, 2019, pp. 2324–2335.

[27] C. Sun, X. Qiu, Y. Xu, and X. Huang, "How to fine-tune BERT for text classification?," in *Proc. China Nat. Conf. Chin. Comput. Linguistics*, 2019, pp. 194–206.

[28] A. Vaswani et al., "Attention is all you need," in *Proc. Adv. Neural Inf. Process. Syst.*, 2017, pp. 5998–6008.

[29] D. Mandic and J. Chambers, *Recurrent Neural Networks for Prediction: Learning Algorithms, Architectures and Stability*. Hoboken, NJ, USA: Wiley, 2001.

[30] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Comput.*, vol. 9, no. 8, pp. 1735–1780, 1997.

[31] J. Keim, A. Kaplan, A. Koziolek, and M. Mirakhorli, "Does BERT understand code?–An exploratory study on the detection of architectural tactics in code," in *Proc. Eur. Conf. Softw. Archit.*, 2020, pp. 220–228.

[32] D. Guo et al., "GraphCodeBERT: Pre-training code representations with data flow," in *Proc. 9th Int. Conf. Learn. Representations*, 2021, pp. 1–18.

[33] A. Kanade, P. Maniatis, G. Balakrishnan, and K. Shi, "Learning and evaluating contextual embedding of source code," in *Proc. Int. Conf. Mach. Learn.*, 2020, pp. 5110–5121.

[34] X. Jiang, Z. Zheng, C. Lyu, L. Li, and L. Lyu, "TreeBERT: A tree-based pre-trained model for programming language," *Proc. 37th Conf. Uncertainty Artif. Intell., Mach. Learn. Res.*, vol. 161, pp. 54–63, 2021.

[35] H. Husain, H.-H. Wu, T. Gazit, M. Allamanis, and M. Brockschmidt, "CodeSearchNet challenge: Evaluating the state of semantic code search," 2019, *arXiv:1909.09436*.

Authorized licensed use limited to: Indian Institute of Information Technology Lucknow. Downloaded on November 20,2024 at 14:05:10 UTC from IEEE Xplore. Restrictions apply.

[36] Y. Wu et al., "Google's neural machine translation system: Bridging the gap between human and machine translation," 2016, *arXiv:1609.08144*.
[37] K. Clark, M.-T. Luong, Q. V. Le, and C. D. Manning, "ELECTRA: Pre-training text encoders as discriminators rather than generators," in *Proc. 8th Int. Conf. Learn. Representations*, 2020, pp. 1–18.
[38] C. Pan, M. Lu, and B. Xu, "An empirical study on software defect prediction using CodeBERT model," *Appl. Sci.*, vol. 11, no. 11, 2021, Art. no. 4793.
[39] J. Wu, "Literature review on vulnerability detection using NLP technology," 2021, *arXiv:2104.11230*.
[40] J. Howard and S. Ruder, "Universal language model fine-tuning for text classification," in *Proc. 56th Annu. Meeting Assoc. Comput. Linguistics*, 2018, pp. 328–339.
[41] A. F. A., "Deep learning using rectified linear units (ReLU)," 2018, *arXiv:1803.08375*.
[42] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: A simple way to prevent neural networks from overfitting," *J. Mach. Learn. Res.*, vol. 15, no. 1, pp. 1929–1958, 2014.
[43] S. El Anigri, M. M. Himmi, and A. Mahmoudi, "How BERT's dropout fine-tuning affects text classification?," in *Proc. Int. Conf. Bus. Intell.*, 2021, pp. 130–139.
[44] Z. Yao, A. Gholami, S. Shen, M. Mustafa, K. Keutzer, and M. Mahoney, "ADAHESSIAN: An adaptive second order optimizer for machine learning," *Proc. AAAI Conf. Artif. Intell.*, vol. 35, no. 12, pp. 10665–10673, 2021.
[45] W. Aljedaani et al., "Test smell detection tools: A systematic mapping study," in *Proc. Eval. Assessment Softw. Eng.*, 2021, pp. 170–180.
[46] A. Peruma, K. Almalki, C. D. Newman, M. W. Mkaouer, A. Ouni, and F. Palomba, "TsDetect: An open source test smells detection tool," in *Proc. 28th ACM Joint Meeting Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng.*, 2020, pp. 1650–1654.
[47] T. Virgínio et al., "JNose: Java test smell detector," in *Proc. 34th Braz. Symp. Softw. Eng.*, 2020, pp. 564–569.
[48] R. E. Noonan, "An algorithm for generating abstract syntax trees," *Comput. Lang.*, vol. 10, no. 3/4, pp. 225–236, 1985.
[49] A. Panichella, S. Panichella, G. Fraser, A. A. Sawant, and V. J. Hellendoorn, "Revisiting test smells in automatically generated tests: Limitations, pitfalls, and opportunities," in *Proc. IEEE Int. Conf. Softw. Maintenance Evol.*, 2020, pp. 523–533.
[50] A. Wei, P. Yi, T. Xie, D. Marinov, and W. Lam, "Probabilistic and systematic coverage of consecutive test-method pairs for detecting order-dependent flaky tests," in *Proc. Int. Conf. Tools Algorithms Construction Anal. Syst.*, 2021, pp. 270–287.
[51] W. Lam, S. Winter, A. Wei, T. Xie, D. Marinov, and J. Bell, "A large-scale longitudinal study of flaky tests," *Proc. ACM Program. Lang.*, vol. 4, no. OOPSLA, pp. 1–29, 2020.
[52] W. Lam, S. Winter, A. Astorga, V. Stodden, and D. Marinov, "Understanding reproducibility and characteristics of flaky tests through test reruns in java projects," in *Proc. IEEE 31st Int. Symp. Softw. Rel. Eng.*, 2020, pp. 403–413.
[53] W. Lam, A. Shi, R. Oei, S. Zhang, M. D. Ernst, and T. Xie, "Dependent-test-aware regression testing techniques," in *Proc. 29th ACM SIGSOFT Int. Symp. Softw. Testing Anal.*, 2020, pp. 298–311.
[54] A. Shi, W. Lam, R. Oei, T. Xie, and D. Marinov, "iFixFlakies: A framework for automatically fixing order-dependent flaky tests," in *Proc. 27th ACM Joint Meeting Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng.*, 2019, pp. 545–555.
[55] Flakify: A Black-Box, Language Model-based Predictor for Flaky Tests – Replication Package, 2022. [Online]. Available: https://doi.org/10.5281/zenodo.6994692
[56] N. V. Chawla, K. W. Bowyer, L. O. Hall, and W. P. Kegelmeyer, "SMOTE: Synthetic minority over-sampling technique," *J. Artif. Intell. Res.*, vol. 16, pp. 321–357, 2002.
[57] P. Branco, L. Torgo, and R. P. Ribeiro, "A survey of predictive modeling on imbalanced domains," *ACM Comput. Surv.*, vol. 49, no. 2, pp. 1–50, 2016.
[58] C. Goutte and E. Gaussier, "A probabilistic interpretation of precision, recall and f-score, with implication for evaluation," in *Proc. Eur. Conf. Inf. Retrieval*, 2005, pp. 345–359.
[59] M. Raymond and F. Rousset, "An exact test for population differentiation," *Evolution*, vol. 49, pp. 1280–1283, 1995.
[60] J. Micco, "Flaky tests at Google and how we mitigate them," 2016. [Online]. Available: https://testing.googleblog.com/2016/05/flaky-tests-at-google-and-how-we.html
[61] A. Memon et al., "Taming Google-scale continuous testing," in *Proc. IEEE/ACM 39th Int. Conf. Softw. Eng.: Softw. Eng. Pract. Track*, 2017, pp. 233–242.
[62] G. Melotti, C. Premebida, J. J. Bird, D. R. Faria, and N. Gonçalves, "Probabilistic object classification using CNN ML-MAP layers," 2020, *arXiv:2005.14565*.
[63] C. Guo, G. Pleiss, Y. Sun, and K. Q. Weinberger, "On calibration of modern neural networks," in *Proc. Int. Conf. Mach. Learn.*, 2017, pp. 1321–1330.
[64] B. H. P. Camara, M. A. G. Silva, A. T. Endo, and S. R. Vergilio, "What is the vocabulary of flaky tests? An extended replication," in *Proc. IEEE/ACM 29th Int. Conf. Prog. Comprehension*, 2021, pp. 444–454.
[65] A. Memon and J. Micco, "How flaky tests in continuous integration," 2016. [Online]. Available: https://www.youtube.com/watch?v=CrzpkF1-VsA
[66] E. Kowalczyk, K. Nair, Z. Gao, L. Silberstein, T. Long, and A. Memon, "Modeling and ranking flaky tests at Apple," in *Proc. IEEE/ACM 42nd Int. Conf. Softw. Eng.: Softw. Eng. Pract.*, 2020, pp. 110–119.
[67] Identifying and analyzing flaky tests in maven and gradle builds, 2019. Accessed: Jan. 11, 2021. [Online]. Available: https://gradle.com/blog/flaky-tests
[68] T. A. Ghaleb, D. A. da Costa, Y. Zou, and A. E. Hassan, "Studying the impact of noises in build breakage data," *IEEE Trans. Softw. Eng.*, vol. 47, no. 09, pp. 1998–2011, Sep. 2021.
[69] J. Lampel, S. Just, S. Apel, and A. Zeller, "When life gives you oranges: Detecting and diagnosing intermittent job failures at mozilla," in *Proc. 29th ACM Joint Meeting Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng.*, 2021, pp. 1381–1392.

Sakina Fatima received the Erasmus Mundus Joint master's degree in dependable software systems from the University of St Andrews, U.K., and Maynooth University, Ireland. She is currently working toward the PhD degree with the School of EECS, University of Ottawa and a member of Nanda Lab. In 2019, she was awarded the French Government Medal and the National University of Ireland prize for distinction in collaborative degrees. Her research interests include automated software testing, natural language processing and applied machine learning.

Taher A. Ghaleb received the BSc degree in information technology from Taiz University, Yemen, in 2008, and the MSc degree in computer science from the King Fahd University of Petroleum and Minerals, Saudi Arabia, in 2016, and the PhD degree in computing from Queen's University, Canada, in 2021. He is a postdoctoral research fellow with the School of EECS, University of Ottawa, Canada. During his PhD, he held an Ontario Trillium Scholarship, a highly prestigious award for doctoral students. He worked as a research/teaching assistant. His research interests include continuous integration, software testing, mining software repositories, applied data science and machine learning, program analysis, and empirical software engineering.

Lionel Briand (Fellow, IEEE) is professor of software engineering and has shared appointments between (1) School of Electrical Engineering and Computer Science, University of Ottawa, Canada and (2) The SnT centre for Security, Reliability, and Trust, University of Luxembourg. He is the head of the SVV Department, SnT Centre and a Canada research chair in Intelligent Software Dependability and Compliance (Tier 1). He received an ERC Advanced Grant, the most prestigious European individual research award, and has conducted applied research in collaboration with industry for more than 25 years, including projects in the automotive, aerospace, manufacturing, financial, and energy domains. He was elevated to the grades of ACM fellow, granted the IEEE Computer Society Harlan Mills Award (2012), the IEEE Reliability Society Engineer-of-the-year Award (2013), and the ACM SIGSOFT Outstanding Research Award (2022) for his work on software verification and testing. His research interests include testing and verification, search-based software engineering, model-driven development, requirements engineering, and empirical software engineering.

▷ **For more information on this or any other computing topic, please visit our Digital Library at** www.computer.org/csdl.

Authorized licensed use limited to: Indian Institute of Information Technology Lucknow. Downloaded on November 20,2024 at 14:05:10 UTC from IEEE Xplore. Restrictions apply.