Master's Thesis

LEVERAGING INFORMATIVE CODE REPRESENTATIONS FOR FLAKY TEST PREDICTION

BLIRONA KERAJ

May 30, 2023

Advisors:

Johannes Lampel	Chair of Software Engineering
Kallistos Weis	Chair of Software Engineering

Examiners:

Prof. Dr. Sven Apel	Chair of Software Engineering
Prof. Dr. Andreas Zeller	CISPA Helmholtz Center for Information Security

Chair of Software Engineering
Saarland Informatics Campus
Saarland University

UNIVERSITÄT
DES
SAARLANDES

Erklärung

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Statement

I hereby confirm that I have written this thesis on my own and that I have not used any other media or materials than the ones referred to in this thesis

Einverständniserklärung

Ich bin damit einverstanden, dass meine (bestandene) Arbeit in beiden Versionen in die Bibliothek der Informatik aufgenommen und damit veröffentlicht wird.

Declaration of Consent

I agree to make both versions of my thesis (with a passing grade) accessible to the public by having them added to the library of the Computer Science Department.

Saarbrücken,_____ _____

(Datum/Date) (Unterschrift/Signature)

ABSTRACT

Flaky tests in software development refer to non-deterministic tests with the potential to inconsistently pass or fail across executions for the same source code version. Their presence compromises the reliability on regression testing and hinders the software development process. The common practice followed to point to these tests is rerunning them multiple times until they manifest into inconsistent results. Besides the fact that rerunning is costly, it is also not directly concerned with flaky test detection but rather with their aftermath which is the intermittent failures. To address this issue, researchers have developed several machine-learning-based tools for flakiness detection. However, these tools lack interpretability and do not focus on providing meaningful insights into the root causes of test flakiness. Additionally, some of them require test execution and access to Code Under Test (CUT).

We develop Path2Flake, an approach that utilizes code representations to extract information from the test snippets. The root cause of flakiness is located in the code statements, hence we use the code as the main guide to classification. The code representations are the intermediary between the test snippets and the flaky test classification model. We represent test cases by their Abstract Syntax Tree (AST) path-contexts. The neural classification model learns the vector embedding of these representations. This encoding transfers rich information from the test cases to the model to discriminate between flaky and non-flaky tests and serves the goal of providing interpretability.

The approach is evaluated on a publicly available dataset, and the results suggest that it is a promising direction of research to work upon. The results demonstrate that path-context code representations are effective in the task of flakiness detection. Even though the evaluation outcome does not provide solid evidence for the ability of the model to point to flakiness root causes, the Path2Flake architecture and the meaningful encodings provide the infrastructure to classify flaky tests and give explanations for the classification. This thesis contribution is a preventive solution for flaky tests that does not require test execution for classification and with the potential to provide developers with useful hints that can be relevant to debugging flakiness.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LISTINGS

ACRONYMS

AST Abstract Syntax Tree
CUT Code Under Test
I/O Input/Output
NLP Natural Language Processing
BCE Binary Cross Entropy
TBCNN Tree-Based Convolutional Neural Network
GPU Graphics Processing Unit

1 INTRODUCTION

Regression testing is a vital aspect of software development. It involves automated tests to validate that implemented changes or bug fixes do not have an adverse impact on the software. Software integrity is crucial to achieving high-velocity delivery of new features [46]. Tests should run reliably to ensure software integrity and point out any regression introduced by recent changes [55]. Due to non-deterministic tests, this is not always the case, and test failures do not always signal code bugs. Such tests are referred to as *flaky tests*. This non-determinism is manifested as inconsistent passing or failing of these tests across executions for the same version of source code [19, 29, 35].

Flaky tests' presence on projects affects the reliability on the test results and the project itself [38]. Non-determinism makes it extremely challenging for flaky test results to be reproduced. This makes debugging very cumbersome and time-consuming [38]. The burden imposed on software development is significant. In a survey conducted at Microsoft by Lam et al. [29] was found that flaky tests are the second most important reason, out of 10 reasons, for slowing down software deployments. Flakiness in tests causes intermittent failures [31]. The prevalence of flaky tests and the negative impact that they cause on the development process, emphasize the necessity for research on this topic [29, 44]. The main topics that have emerged in this research domain are the analysis of the root causes and the detection of flaky tests.

Flakiness detection is concerned with identifying tests that exhibit non-deterministic behavior with the potential to inconsistently fail in the test suite. The most common approach used when encountering failures from flaky tests is rerunning. This approach attempts to unveil the inconsistency in test results, by running the test multiple times on the same source code version [9, 12, 23, 30]. Rerunning failing tests is a prevalent practice in the industry, where developers persistently repeat the execution with the hope that the test will eventually pass [37, 38]. However, this practice is very inefficient and costly [12, 38]. Notably, Google spends between 2% and 16% of the test execution budget on running flaky tests. Moreover, rerunning tests is merely a temporary fix that can hide code bugs. Unveiling flakiness can be challenging, even after numerous reruns, and these repeated executions impose substantial computational expenses [9].

Motivated to develop more efficient detection techniques, researchers implemented several tools with different underlying approaches most of which are machine-learning-based [9, 20, 44]. Some of the approaches require access to Code Under Test (CUT), or manually predefine the set of features. The meaningfulness and explanatory nature of the input used by these tools is overlooked, making them weak in terms of interpretability. These tools leverage information from test cases in different ways. Some of them rely on textual information, meaning that they learn on the surface of the test snippets by applying Natural Language Processing (NLP) techniques [9, 20, 44]. Others incorporate test rerunning logs and dynamic instrumentation of test execution in their analysis to detect flakiness [12, 29, 35].

An important component of a successful machine learning model is the input features from the training data [27]. The representation of code is a critical factor in the performance of machine learning models that are trained on source code [2, 7, 8, 24]. In order to effectively utilize information from the test code for teaching the model to perform flakiness detection, we should extract informative and meaningful code representations. Different techniques can be used to represent code, such as tokenization, AST, or high-level graph-based representations [2, 7, 57].

We present *Path2Flake*, an approach that leverages informative and meaningful code representations to perform interpretable classifications of flaky tests. To that end, we implemented a predictor that aims to indicate whether and why a test is flaky. We use AST path-contexts as code representations because they allow us to exploit the structured nature of the code [2, 7]. Our work is an attempt towards providing preventive solutions for flaky tests. As opposed to some of the currently developed flaky test predictors, we identify such tests before executing them to avoid their adverse manifestation. As such, *Path2Flake* is a static approach, meaning that it does not require access to the CUT or test execution to do the prediction. This is motivated by one of the study implications conducted by Luo et al. [35], who state that most of the flaky tests are flaky the first time they are written, hence we should develop techniques that detect such tests without executing them. We also focus on reducing the debugging efforts by providing the developers with useful information for the flakiness outcome label. The encoding of test cases coupled with the architecture of the neural attention model serve the goal of giving implicit information about the root cause of flakiness [53].

We evaluated *Path2Flake* on a publicly available dataset labeled by Alshammari et al. [9]. This dataset contains 20,783 test cases as part of 24 Java projects. The results indicate that path-contexts as test case representations are effective in the task of flaky test classification. This is evident from the performance results of Path2Flake compared to the baseline classifiers and to the state-of-the-art. Such results are evidence of a promising approach in the direction of interpretable flaky test classifiers. As the initial endeavor in this direction, Path2Flake establishes the necessary framework for classifying flaky tests and offering explanatory insights into the classification.

2 BACKGROUND

In the following sections, we briefly discuss flaky tests and their root causes. Given that the objective of this thesis is to offer a preventive solution by detecting flakiness, it is crucial to differentiate between flaky tests and intermittent failures. Additionally, we will delve into the topic of code representation, as it is fundamental to our approach in combination with the attention neural model.

2.1 FLAKY TESTS

Flaky tests are tests that can exhibit non-deterministic results, i.e. the outcome may not be consistent across multiple executions despite the unchanged code. Luo et al. [35] define flaky tests as those that can intermittently pass or fail, for the same code version. In this definition, there is no regard to the executing environment, which according to Fowler [21] can affect the manifestation of the flakiness. Eck et al. [19] uses the following definition: "Software tests are flaky when they exhibit a seemingly random outcome despite exercising code that has not been changed."

Flaky tests compromise the purpose of the test suites that validate the software changes and that make sure no existing functionality is broken. With such unreliability introduced to testing, there is an increase in effort and time to diagnose issues that do not exist or to overlook actual bugs [21]. Additionally, such tests are prevalent, especially on large codebases. At Google, a daily average of 73K out of 1.6M (4.56%) test failures were caused by flaky tests [38]. As a result, test flakiness has emerged as an important research topic.

2.1.1 *Intermittent failures vs Flaky Tests*

It is important to make the distinction between flaky tests and intermittent failures. Intermittent failures are job executions that pass and fail at least once under the same configuration during software testing or system operation [21, 31]. Failures caused by the potential of flaky tests to fail inconsistently are called intermittent failures [31, 51]. These failures are the manifestation of flaky tests. However, flaky tests are not the single reason for intermittent failures to occur, as other underlying infrastructure issues may be the cause [31]. Intermittent failures and flaky tests are related concepts but are not identical.

Intermittent failures are typically caused by environmental factors, such as race conditions or system load, that affect the behavior of the system under test [49]. Flaky tests, on the other hand, are usually caused by issues in the test code, such as timing or synchronization problems, that introduce non-deterministic behavior [29, 35, 44].

Tufano et al. [51] differentiate between intermittent failures and flaky tests, with intermittent failures being characterized by being unreproducible or sporadic failures that can be eventually fixed with a small number of modifications, and flaky tests being characterized by producing inconsistent results due to some kind of non-determinism in the test code.

Addressing intermittent failures typically involves identifying and resolving environmental issues that affect the system under test, while addressing flaky tests typically involves refactoring the test code to eliminate sources of non-deterministic behavior [49].

Flaky tests can fail due to their flakiness manifesting as intermittent failures but also due to code bugs resulting in legitimate failures. In such cases, many legitimate failures are ignored as a result of high level of false positives [38]. We aim to find a preventive solution, hence we focus on the detection of the test flakiness itself before the failure occurs.

2.1.2 *Flaky tests Root Causes*

The widespread presence of flaky tests in projects, especially large-scale ones, has a significant impact on productivity and computational resources [31, 38]. Even if the number of distinct flaky tests is not very high, the number of failed builds they can cause is significant [29]. This fact motivates many research studies done on flaky tests, specifically on their root causes. However, such tests are very difficult to investigate as flakiness is challenging to reproduce [29].

Researchers have pinpointed some root causes that introduce flakiness in tests [29, 35, 44]. The categories of these root causes can overlap with each-other. Luo et al. [35] analyze the commit history of all projects from the Apache Software Foundation. After the filtering phase, they end up with 486 commits that perform flaky test fixes and investigate 201 out of them to define the root causes of flakiness. One of the prominent causes of flaky tests are ***asynchronous waits***. Such a scenario occurs when the test performs asynchronous calls whose output is crucial for the rest of the test execution and is not available when needed to be used. These calls being asynchronous, are not always quickly responsive. An example of such a flaky test is the test snippet in Listing 2.1. This test case is part of the HBase project in the dataset analyzed by Luo et al. [35]. This test starts a server `firstServer` and waits for it to ping back using `Thread.sleep(2000)`. If the response is not received on time, the subsequent assertion fails. Tests that rely on such calls are flaky, with the potential to fail and pass inconsistently. In the study of Luo et al. [35], 46% of commits belong to this category.

Listing 2.1: A flaky test caused by asynchronous waits

```
@Test
 public void testRsReportsWrongServerName() throws Exception {
 MiniHBaseClusterRegionServer firstServer =cluster.getRegionServer(0);
 firstServer.setHServerInfo();

 // Sleep while the region server pings back
 Thread.sleep(2000);
 assertTrue(firstServer.isOnline());
  ... // similarly for secondServer
}
```

Another prevalent root cause of flakiness is ***concurrency*** [29, 35]. In this case, flakiness stems from the non-deterministic interaction of threads, examples of which are deadlocks

and data races. Several threads may attempt to simultaneously perform modification on a source, and this may cause the manifestation of flakiness. A possible solution is ensuring synchronization and making threads execute atomically. In the study of Luo et al. [35], 20% of commits belong to this category.

In many test cases, operations that rely on randomly generated numbers are frequently utilized [29]. An example of this kind of flaky test, from Microsoft's test suite, is presented in Listing 2.2. This test instantiates a random object without a seed, which implies the default seed to be the system time. Executing `Random().Next()` consecutively may generate the same `id` [29]. The unpredictable nature of ***random values*** can result in tests either passing or failing. The timestamp at which the random generation occurs can be used as a seed. If the interval between generation of values is very small, the seed will not change causing the test to potentially fail. ***System time*** is also another source of flakiness where granularity and timezone changes are the most prominent ones [29].

In addition, flaky tests were found to be associated with other factors, such as ***test smells*** [16, 42]. Test smells are defined as poor design and implementation practices, whose presence may negatively affect comprehension and maintenance of test suites [18, 51]. Such smells are: *duplicate test code* leading to inconsistencies in test results, as changes to one copy of the test may not be reflected in the other copies; *test code fragility*, overly dependent on the implementation details of the system under test can break easily when those details change; *assertion roulette*, presence of more than one assertion, so if one may fail it is difficult to define which one; *sleepy test*, rely on delays invoking the thread.sleep() method to accommodate asynchronous operations, etc.

Listing 2.2: A flaky test caused by random values

```
class TestAlertTest {
 void TestUnhandledItemsWithFilters() {
  TestAlert ta1 = CreateTestAlert();
  TestAlert ta2 = CreateTestAlert();
  ...
  Assert.AreNotEquals(ta1.TestID, ta2.TestID);
  }
  TestAlert CreateTestAlert() {
  //random object that introduces flakiness in the test
  int id = new Random().Next();
  ...
  return new TestAlert(TestID = id, ...);
  }
 }
```

These and other root causes, such as ***resource leakage, test order dependency, network, Input/Output (I/O) operations, floating point operations***, are the focus of researchers and developers to find ways to easily identify and tame them [35].

All these root causes are mostly concerned with the test code itself rather than Code Under Test (CUT). Additionally, access to the production code is not always easy to obtain. Hence, flakiness detectors that include CUT in the analysis have issues with scalability and

lack of permission to access CUT [20]. Hence, we focus on the static analysis of flaky tests, rather than the dynamic one.

2.2 CODE REPRESENTATIONS

The amount of source code available in large repositories makes it a great subject of study for software engineering applications. Additionally, alongside source code, there is a great amount of metadata related to it that explains changes, bug fixes, and code reviews [2]. Learning from these repositories can help in leveraging many good programming practices, bug-fixes, to transfer knowledge into recommendation systems, code analysis tools, flakiness detection, and other tasks [2, 4, 8, 56]. Writing and maintaining source code is a crucial activity for programmers and developers. Since the source code on which is operated is huge, the costs are also considerable [5]. The availability of "big code" suggests a new, data-driven approach to tackle such issues [2]. A promising option is machine learning, whose power stands in the ability to generalize from examples and handle noise [22]. It has already been used in many applications to replace heuristics and manual feature engineering.

One of the essential aspects of a satisfactory machine learning model is the training data [27]. In this case, source code is the data fed into the model for performing particular tasks. To effectively utilize the information embedded in the source code, it is essential to supply the model with informative and meaningful code representations [2, 4]. Code representations are the intermediary between the machine learning model and the source code used for training [52]. Many researchers have developed approaches for extracting code representation ranging from the ones based on token streams to those based on data and control flows.

2.2.1 *Natural Language vs Programming Language*

Source code has been handled as natural language text by being exploited through the lenses of NLP techniques. The main argument behind this approach is the naturalness hypothesis of code, formulated by Hindle et al. [25], according to which: *Software is a form of human communication; software corpora have similar statistical properties to natural language corpora; and these properties can be exploited to build better software engineering tools.* Continuing this line of thought, it is reasonable to say that code has rich patterns similar to natural language that can be exploited using NLP techniques. This unlocks the possibility to learn how developers naturally write code, introducing development in code generating domain. As being written by programmers using conventions, idioms, and readable style, code is more *predictable*.

Viewing source code from this perspective encourages learning program properties on the surface of the code snippets [6, 8]. However, as much as it is similar to natural language, source code has clear differences. Code is executable and as such, it has formal syntax and semantics [2]. It may be a "medium" of communication between programmers but code should be comprehensible by the machine as well. This makes code not as flexible as natural language. Formal syntax and semantics impose a structural nature on code which encodes many program properties [8]. This structure and difference to natural language should be leveraged in order to ensure effective learning on the source code. Code representations

should be able to capture this structure in order to transfer to the machine learning model as much information as possible [7].

2.3 NEURAL ATTENTION MODEL

Attention mechanisms in neural models are a powerful technique, allowing networks to selectively focus on different parts of the input data when making predictions or generating output [53]. As such, the attention mechanism provides interpretability and also a more effective learning process. Interpretability stems from the weights that the attention mechanism assigns for each input, which are learned during training [8, 39]. These weights encode the contribution each input had in calculating the model's classification output. Also, being able to focus on relevant input for the particular classification task, guides the model to learn from informative input, making it more effective [52].

Consider the input vectors $\mathbf{x_0}$, $\mathbf{x_1}$, ..., $\mathbf{x_n}$ to the attention layer. Attention vector $\mathbf{a}$ is made of attention weights α which factor the inputs in a linear combination:

$$\mathbf{y} = \sum_{i=1}^{n} \alpha_i \cdot \mathbf{x_i}$$

This linear combination produces a single output y, used further in classification. Given attention vector $\mathbf{a}$ and input values, individual weights can be retrieved as:

$$\alpha_i = \frac{\exp(\mathbf{x_i^T} \cdot \mathbf{a})}{\sum_{j=1}^{n} \exp(\mathbf{x_j^T} \cdot \mathbf{a})}$$

This attention mechanism was introduced by Luong et al. [36], and we use it to aggregate the AST path-context vectors into a single code vector representation, which is the input to the flakiness prediction layer of the model. Attention weights identify the paths in code that are most informative in detecting flakiness.

2.4 OPTIMIZATION ALGORITHM

Optimization algorithms are used to find the optimal set of parameters that result in the best model performance. One of the widely used optimization algorithm is `Adam` (Adaptive Moment Estimation). This algorithm, introduced by Diederik P. Kingma and Jimmy Ba [28], is part of the optimization approaches with adaptive learning rates. The learning rate is a network hyperparameter that controls the step of change applied to the model parameters to reach convergence and find the minimum of the loss function. Large learning rates lead faster to the minimum but may never reach it due to the big steps it takes. Small learning rates, take smaller steps to reach convergence, but results in a longer training time and are prone to stuck in local minima [14]. `Adam` computes adaptive learning rates for each parameter by taking into account the first and second moments of the gradients [28]. The first moment is the average of the gradients, while the second moment is the average of the squared gradients. The algorithm uses these moments to calculate an exponential moving average of the gradients, which is used to update the parameters.

This optimization approach is formalized by Goodfellow et al. in their book *"Deep Learning"* [22], as below:

Algorithm 1 (Adam)

Require: *Step size* ϵ
Require: *Exponential decay rates for moment estimates,* ρ_1 *and* ρ_2 *in* $[0, 1)$
Require: *Small constant* γ *used for numerical stabilization (default :* 10^{-8}*)*
Require: *Parameters* θ
Initialize 1st and 2nd moment variables $s = 0, r = 0$
Initialize time step $t = 0$

while *stopping criterion not met do*
  Sample a minibatch of m examples from the training set $x^1, ..., x^m$ *with corresponding labels* y^i
  Compute gradient: $g = \frac{1}{m} \nabla_\theta \sum_i (\mathcal{L}(x^i; \theta), y^i)$
  $t = t + 1$
  Update biased first moment estimate: $s = \rho_1 s + (1 - \rho_1) g$
  Update biased second moment estimate: $r = \rho_2 r + (1 - \rho_2) g \cdot g$
  Correct bias in first moment: $\hat{s} = \frac{s}{1 - \rho_1^t}$
  Correct bias in second moment: $\hat{r} = \frac{r}{1 - \rho_2^t}$
  Compute update: $\nabla\theta = -\epsilon \frac{\hat{s}}{\sqrt{\hat{r}} + \gamma}$
  Update parameter: $\theta = \theta + \nabla\theta$
end while

The first and second moments allow the optimizer to update the parameters based on the exponentially decaying moving average of recent gradients. This way the convergence is reached faster and the very past gradients effect is faded. The bias corrections are performed to discard the bias introduced by the ρ_1 and ρ_2 initialization. The hyperparameters ρ_1 and ρ_2 control the decay rates of the moving averages. According to the authors Diederik P. Kingma and Jimmy Ba [28] these hyperparameters need no tunning, however, it is advised [40] to tune the initial learning rate[1].

1 Hyperparameter tunning is explained in Section 5.3.2

3 RELATED WORK

Several researchers and practitioners have studied the recurring problem of flakiness in regression testing. The existing work in this area can be broadly classified into two categories: the first being related to the analysis of root causes, and the second related to flakiness detection. One potential solution to detecting flaky tests is to leverage informative code representation approaches. Machine learning-based approaches that learn from source code use mainly one of the following code representation approaches: tokenization, tree-based representations, and high-level graphs.

3.1 ROOT CAUSE ANALYSIS

The first extensive study on flaky tests is done by Luo et al. [35]. The authors analyze in detail 201 commits that likely fix flaky tests from 51 open-source projects from `the Apache Software Foundation`. The outcome of this analysis is the list of common causes of flakiness, attempts for detection, and fixing strategies. Asynchronous Wait, Concurrency, and Test Order Dependency are three of the ten root causes that account for 77% of all commits [35]. This study includes code examples of flakiness for each cause, which provides some root-specific patterns.

Another study on the same topic was conducted by Lam et al. [29] whose results were similar to Luo et al. [35]. This study provides a tool called `RootFinder` that analyzes the logs of passing and failing executions of the same test to suggest method calls that could be responsible for the flakiness. This tool is based on `Torch Instrumentation`, which produces instrumented versions of all of the tests' dependencies, by logging various runtime properties. These instrumented versions of tests are run 100 times in an attempt to produce logs for both passing and failing executions, which are analyzed by `RootFinder` to point out the cause of flakiness.

Other research has established a relationship between test smells and flaky tests [11, 51]. In the study of Palomba and Zaidman [42], refactoring the tests fixes the flakiness in all the tests containing the code smell. As flakiness may originate from test smells, such studies motivate our own study of building code representations that capture these test smells.

3.2 DETECTING FLAKY TESTS

The commonly used approach to point to flaky tests which is concerned with their manifestation as intermittent failures, is rerunning. Failing tests are rerun multiple times, to examine whether a test failure is caused by a newly introduced bug or by the presence of flakiness in the test [35]. This approach is not directly addressing flakiness but rather its manifestation as an intermittent failure. A flaky test that rarely exhibits its flakiness makes the practice of rerunning ineffective. Also, it requires significant computational resources [12, 38]. `DeFlaker` [12] is a tool that monitors code coverage of the test cases and marks as

flaky only those tests that failed while running on unchanged code. Similar tools that use the rerunning approach are `iDFlakies` [30] and `NonDex` [23].

Due to their inefficiency, work has been done to replace rerunning-based tools with more sophisticated ones, which are predominantly machine learning-based. As mentioned in the previous section, test smells are used as indicators for flakiness in tests. Camara et al. [16] proposed an approach for predicting test flakiness using test smells as prediction features. This tool dynamically gathers information about test execution and includes the production code in its analysis. Sometimes access to production code is not easy to obtain [20]. This hindrance and the limited scalability circumscribe the use cases of such dynamic tools.

One of the detectors, part of the static detection tools family, is the vocabulary-based approach [44]. Pinto et al. [44] concluded that flaky tests seem to follow a set of syntactical patterns. Hence, this fact was utilized by implementing the NLP techniques to predict the flakiness of a test even before running it and without accessing CUT. In order to employ NLP techniques for prediction, tokens such as identifiers and method names are extracted from the test code. After some preprocessing steps, these tokens serve as input for the machine learning algorithm.

This work aimed at an automated lexical analysis of test cases and inspired the development of `FlakeFlagger` by Alshamari et al. [9] with some improvements and corrections. Alshammari et al. [9] proposed to predict flaky tests using dynamically computed features capturing code coverage, execution history, and test smells. This study shows that some of the features used by Pinto et al. [44] on the vocabulary approach were not useful in identifying flakiness, as the occurrence of these features on flaky tests was only a pattern of the specific projects used in that study.

`Flakify` [20] is a black box classifier that uses the `CodeBERT` 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. Hence, `Flakify` does not require a pre-definition of features to be used as predictors for flaky tests. It is a neural network made from a multi-layer bidirectional transformer. The complex way data is processed and the many data vector aggregations lose the interpretability, hence the name black-box [34]. Our work goes in the opposite direction, where we try to explain the results through code representations' contribution to outcome calculation. `Flakify` may have high accuracy but giving up a small tolerable percentage of it can provide valuable explainability.

3.3 CODE REPRESENTATIONS

Many software engineering tasks are aided by the introduction of machine learning and deep learning. Source code is subject of exploration for a wide spectrum of program analysis tasks. Researchers have developed approaches for extracting code representation ranging from the ones based on token streams to those based on data flows.

3.3.1 *Code as a Stream of Tokens*

In many existing approaches, NLP methods are applied to source code to get code representations. They are built on top of the conjecture that most software code is also natural like languages, in the sense that it is created by humans, and thus, like natural language,

it is also likely to be repetitive and predictable [25]. `Flakify` predictor, mentioned in the previous section, uses CodeBERT to construct informative code representation to learn the correspondence between flakiness and patterns in test cases [20].

Pinto et al. [44] in an attempt to define the vocabulary of the flaky tests, use tokens as code representations on which they apply the bag-of-words naive method [44, 58]. `FlakeFlagger` also is a predictor that learns vector embeddings for code tokens, among other features, to perform flakiness classification. Source code is treated as text, even in many automated code review studies, beyond the realm of flakiness classification. One example is work done by Allamanis et al. [3] and Iyer et al. [26], whose goal is to perform extreme code summarization by pushing sub-tokenized code snippets into a convolutional attention model. Code tokens are leveraged to find important locations in source code by extracting position and context-dependent features of tokens.

3.3.2 *Tree Based Representations*

NLP techniques have the major drawback of not being able to capture and use the structured nature of code. Attempting to overcome this issue and to build more informative code representations, many authors have used abstract syntax trees. AST representations can capture regularities and code patterns in a more effective and efficient manner compared to tokens.

`Code2vec` is one of many tools that embrace the structured nature of code and learn vector embeddings from the AST path representations of code snippet [8]. Such representations are used as input for a neural network that performs automatic code review, specifically, method naming. The paths extracted from the AST transfer meaningful syntactical information and latent semantics to the network. Our approach is mainly inspired by this work and adapted for the flakiness detection task.

There is also an analogous bimodal approach to `CodeBERT` but using tree-based representation rather than the token stream of code [5]. This model is defined based on the assumption that code is conditional upon language. The code snippet component of the bimodal data is represented by the parse trees and partial parse trees. Terminal nodes are explicit occurrences in the code snippet. The application of this approach is performing retrieval tasks, such as retrieving source code given natural language input and vice-versa.

Programmers write code not simply to be executed by a computer, but also to communicate to later developers who will adapt, update, test, and maintain the code [2, 4]. Such code is called idiomatic. Idioms themselves are syntactic patterns that encapsulate a certain function or information and occur frequently. `HAGGIS` is a tool that mines source code to uncover interesting code patterns specifically idioms [4]. The code representation the authors use is a parse tree. They use parse trees as the component, the importance of which is measured, in order to output idioms.

Tree-based representations are used as input for TBCNN. Mou et al. [39] encode AST nodes as vector embeddings and introduce the notion of the continuous binary tree which overcomes the problem of the varying number of children in the AST depending on the code

snippet length. The use case of this network is program classification based on functionality and pattern detection. This approach is visualized in a high-level scheme in Figure 3.1[1].

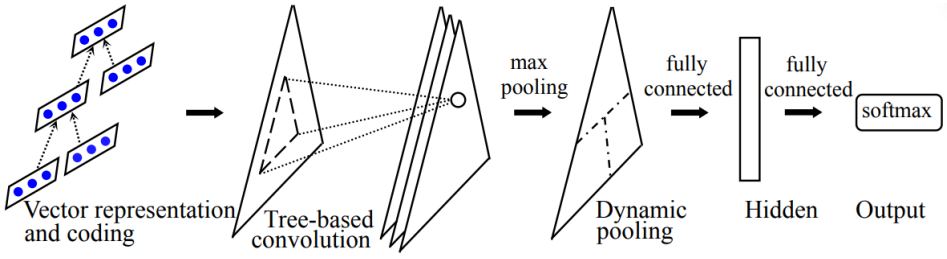


Figure 3.1: The architecture of the TBCNN. The main components in their model include vector representation and coding of the nodes, tree-based convolution, and dynamic pooling. Then the classification is performed by a fully-connected hidden layer and an output layer with softmax activation function [39].

3.3.3 *Graph Code Representations*

There are two signal sources in source code that cannot be captured by previously described code representations: data flow and type hierarchies [57].

Graphs can be a suitable way to represent code components in situations where distant dependencies need to be taken into account for certain tasks. Zhang et al. [57] use graph code representations as input in a Graph Neural Network, that identifies variable misuse and performs variable naming. The process of constructing a graph of source code is based on the AST of the code. Nodes of AST are connected with *child* edges and *next token* edges to represent the hierarchy of the nodes. There are also edges that capture the data flow in code, namely, *last read* and *last write* edges which keep track of uses and updates of terminal nodes representing variables. These and many other edges, ensure that rich information on code semantics is captured by the graph code representation.

The domain of security also utilizes graph-based representations of code. Aiming to find vulnerabilities, Yamaguchi et al. [56] use a comprehensive graph representation of source code, called code property graph. Code property graph is a merge between AST, control flow graphs, and program dependency graphs. This representation is exploited and inspected by graph traversal methods. The large number of code properties captured by this representation enables the development of specific vulnerability-catching templates, that are able to efficiently work on a large amount of source code.

1 This figure is taken from the paper "Convolutional Neural Networks over Tree Structures for Programming Language Processing" by Mou et al. [39]

4

APPROACH

In this chapter, we describe in detail the approach we follow to build Path2Flake. We focus on the code representations extracted from the test cases, which serve as input features for the classification model. Additionally, we will explain the model's architecture and the main operations that take place during model training and classification.

4.1 TEST CASE DECOMPOSITION

The aim of this thesis is to detect the flakiness of a test case using machine learning and provide an explanation for the model's outcome. Predicting a program property given a code snippet requires a meaningful and informative encoding of the snippet, which is the input to the machine learning model. This encoding, also called code representation, should aid effective learning and also augment the interpretability of the model. As already described, there are various code representation options, such as a stream of tokens, parse trees, data flow graphs, control flow graphs, and the like. In deciding how we will represent the test cases, we consider the following trade-off between program analysis and learning effort.

4.1.1 *Program Analysis and Learning Effort Trade-off*

Deciding on the code representation that we use to train the machine learning model for detecting flakiness in tests is very important. The code should be represented in a way that facilitates effective and efficient learning [2]. The model should be able to provide accurate classifications and to generalize to unseen code snippets [22]. To that end, the test case representations should contain sufficient information to encode semantics that hint at the possible flakiness root causes. Additionally, representations should be granular enough to avoid sparsity and overfitting [7]. Sparsity is the problem of having unique and complex input, that does not occur frequently enough among the data points [8]. Its presence obstructs the model's ability to find a relationship between test cases and flakiness. Code representations should also be interpretable pointers in test cases, explaining the model's output label.

Source code has been handled like text in many cases by applying NLP techniques that are developed for natural language text. This approach is motivated by the argument that code, just like natural language, is written by humans and as such, it has regularities that can be exploited by NLP techniques [2, 25]. An example is learning on the surface of source code, by extracting all tokens from a code snippet and using these token streams as code representations.

However, source code has a rich syntactic structure, as opposed to text which is a flat sequence of tokens [2, 7, 25]. In this case, feeding code tokens as input would require the model to learn any syntactic and semantic regularities from scratch, as they are not encoded

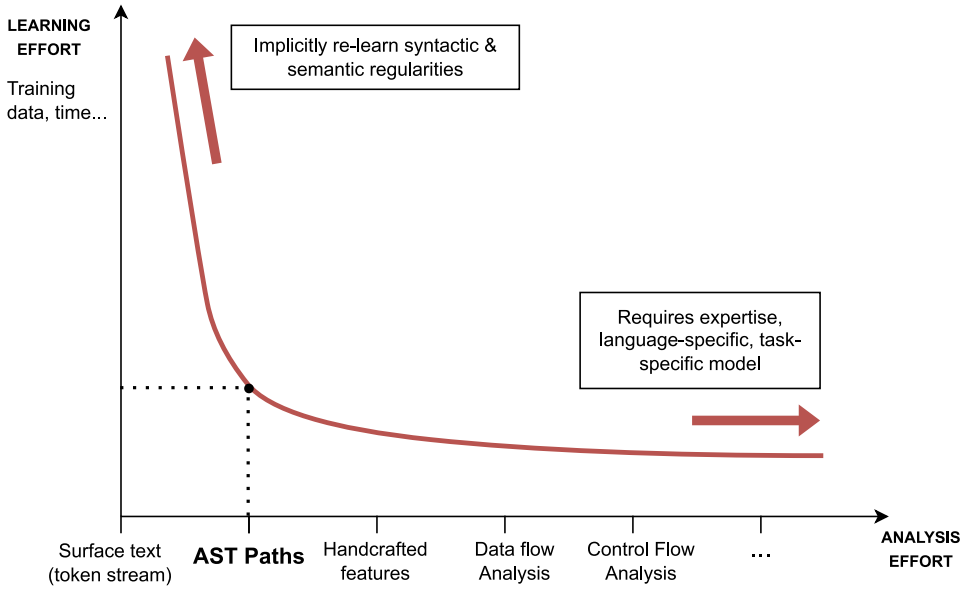


Figure 4.1: Learning effort - Program Analysis effort trade-off [8]: as the graph line indicates, the further in the right of the x-axis the code representation is positioned, the higher the analysis effort they require to be constructed. This analysis effort increase corresponds to the inverse change of the learning effort, which decreases when using these code representations.

in such representations. This demands a high learning effort but a low program analysis for extracting representations [8].

Alternatively, code representations can be built using data or control graphs as in [56, 57]. The structure and syntax of a programming language are preserved in graphs, which makes these representations very informative. However, the program analysis demanded is extensive, as building data or control graphs out of code snippets is computationally expensive and has a tendency to cause overfitting [7, 8].

A commonly employed method for extracting code representations from snippets is the use of Abstract Syntax Tree (AST) paths, which is a technique embraced by the authors of these studies [4–6, 8, 39]. AST paths are the middle ground in this trade-off as they encode syntactical information of the code, lowering the learning effort. Additionally, they are easy to extract and avoid sparsity by decomposing the snippet into paths that can occur among other snippets [8]. The way code representations are positioned in the scope of learning and analysis effort, is visualized in Figure 4.1[1].

Examining the trade-off and aligning it with our objective of predicting flakiness in test cases, we represent the test snippets using AST path-contexts. This choice accommodates the need for effective learning and for interpretability and is based on the approach used by Alon et al. in [7, 8]. By representing a code snippet using its syntactic paths, we can capture regularities that reflect common code patterns [8]. We make such a choice for code representation, assuming that the model learns a correspondence between flakiness and AST path patterns that hint at this property. We conjecture that AST paths are meaningful

1 The figure is from the presentation of code2vec by Alon et al. [8] in the conference of the 46th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL 2019).

enough to encode flakiness root causes and teach the model to discriminate between flaky and non-flaky tests. AST paths coupled with an attention mechanism are utilized to provide a developer with an informative explanation of why Path2Flake produced a particular flakiness label.

4.1.2 *AST path-contexts as test case representations*

Each test case is decomposed into AST paths. These AST paths are made of tree syntactic paths between every pair of elements in the test case. To decompose the test snippet we first build the AST for each test case, whose definition is the following:

Definition 1 (AST)

An Abstract Syntax Tree is a tree representation of the syntactical structure of source code written in a programming language. Nodes in an AST represent various syntactic elements of the code, where non-terminal nodes represent the statements or operators, and the terminal nodes represent the operands or variables, which are token occurrences in the test snippet.

First, the test case is parsed to build the AST. An example of an AST is illustrated in Figure 4.2, which is built by parsing the test snippet in Listing 4.1. We traverse and visit the nodes to extract the syntactic paths, defined as:

Definition 2 (AST path)

An AST path is a path between nodes in the AST that begins at a terminal node n_1, traverses through a sequence of intermediate non-terminal nodes n_2, ..., n_k, and concludes at another terminal node n_{k+1}. Along with these nodes, there is also a direction specification, either up or down, describing how the path navigates in the tree. So, AST path is defined as the sequence:

$n_1 d_1, ..., n_k d_k n_{k+1}$, *where* $n_1, n_{k+1} \in$ *terminal nodes*, $n_2, ..., n_k \in$ *non-terminal nodes*, $d_i \in \{\uparrow, \downarrow\}$.

AST paths have two important properties: *path width* and *path length*. The horizontal distance of two nodes with an immediate mutual parent is defined by the path width. The path length is the number of intermediate non-terminal nodes that make up the path from one terminal node to the other. These properties guide the level of complexity encoded by a code representation. From the definition of AST paths we derive:

Definition 3 (AST path-context)

An AST path-context is a tuple $< t_1, p, t_{k+1} >$, containing the starting terminal node token t_1, the AST path p itself, and the ending terminal node token t_{k+1}. The tokens are associated with the AST terminal nodes n_1 and n_{k+1} respectively. So, AST path-context is defined as:

$$< t_1, (n_1 d_1, ..., n_k d_k n_{k+1}), t_{k+1} >$$

By using AST path contexts, our model can learn rich representations of code by capturing not only the relationships between programming constructs but also their relative positions within the code structure. This is important because the position of a construct in the code can impact its meaning or role. Such contextual information is particularly important in detecting flaky tests, as the order of the statements inside the test can affect the flakiness

Listing 4.1: A flaky test caused by asynchronous waits

```
@Deployment public void testAsyncTask(){
  waitJobs(5000L,200L);
 assertEquals(0,managementService.createJobQuery().count());
}
```

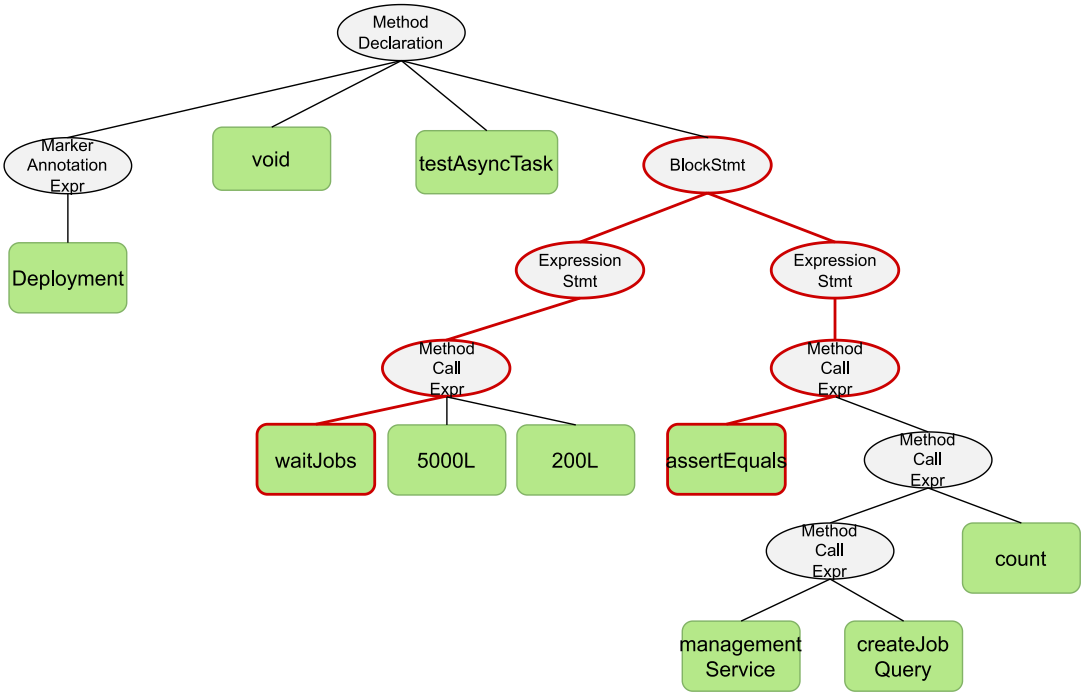


Figure 4.2: AST of a test case snippet

manifestation. By incorporating AST path-contexts, models can better understand these positional differences and their implications on the code's behavior.

4.2 BUILDING CODE REPRESENTATIONS

After we determine the test case representation to use, we continue with describing the process of building and extracting them. We explain how to transform the test snippets into machine-learning comprehensible inputs.

4.2.1 *Path extraction*

The first step in extracting code representations out of test cases is **parsing**. We use flaky and non-flaky tests written in Java, as input to a parsing process, which builds ASTs for each test case. In Figure 4.2 we show the AST of the test snippet from Listing 4.1. Terminal nodes, which are marked in green, are occurrences in the test snippet. Non-terminal nodes

are marked in grey and represent the language constructs such as; statements, loops, expressions, declarations, and the like.

The process of extracting all possible syntactic paths between all pairs of test case elements involves **traversing** the AST. These paths together with terminal node values build the path-contexts, which are used to represent the test case.

Example. In Figure 4.2 we have highlighted an AST path-context between the terminal tokens `waitJobs` and `assertEquals`:

```
<waitJobs,(MethodCallExpr↑ExpressionStmt↑BlockStmt↓ExpressionStmt↓MethodCallExpr),assertEquals>
```

We extract every path-context between pairwise terminal tokens in the test case. Depending on the length of the test cases, the number of path contexts per test case varies. Similarly, the path width and length also vary based on the position and distance between terminal nodes. To set an exploration and complexity limit we predefine: *maximum_path_length* and *maximum_path_width*. By limiting the width, we restrict the exploration of the tree to more closely related sibling nodes and avoid traversing unrelated branches. The *maximum_path_length* sets a limit on the number of non-terminal nodes included in the path to improve the model's generalization ability and reduce noise (long paths may contain irrelevant and noisy information). We set the values 8 and 2 respectively for *maximum_path_length* and *maximum_path_width*. The choice for such values is based on the nature of the flakiness root causes, which is manifested as short statements in the test code and on the common choices in the literature that motivates our work [8, 39].

Additionally, we also set the *path_contexts_per_test_case*, a value that specifies how many path-contexts are going to represent a single test case. This parameter is set to 50, given that the test cases have short lengths and only a few path-contexts can be related to the flakiness root cause. The means by which we ensure that every test case is represented by a predefined number of path-contexts are **padding** and **path sampling** [8].

Padding appends blank spaces if the number of path contexts extracted from a test case is lower than the predefined number of path-contexts per test case. Path sampling randomly selects path contexts out of all extracted from the test case, when the number of the latter exceeds the number of path-contexts per test case. More formally:

Let k be the *path_contexts_per_test_case*, and $path\_context_1$, ..., $path\_context_n$ be the sequence of all AST path-contexts extracted from a test case. By conditioning on k:

if $n < k$, we append $max - k$ blanks spaces to the initial sequence, to build:

$$path\_context_1, ..., path\_context_n, blank\_space_{n+1}, ..., blank\_space_k$$

if $n > k$, we randomly sample k elements from the initial sequence, to build:

$$path\_context_1, ..., path\_context_k$$

We note that these steps of the *Path2Flake* might hinder the learning process since path-contexts, that are relevant in learning a relationship between flakiness and path-context patterns, can be discarded.

Along with the process of building path-contexts, we also count the frequencies of terminal node tokens and path tokens that occur in the input dataset of test cases. These

frequencies are saved into a *dictionary* file which is used to later build a *vocabulary* for the path-contexts. The *path vocabulary* and *token vocabulary* hold all paths that connect them and all terminal node tokens, alongside their indices. These indices are specified based on the sort of their frequencies in the dictionary.

At the end of path extraction and preprocessing, we get a labeled dataset. Each row on the dataset has a flakiness label and the path-contexts for the particular test case. Additionally, we also get a dictionary, with the token and path frequencies, and vocabulary that defines indices.

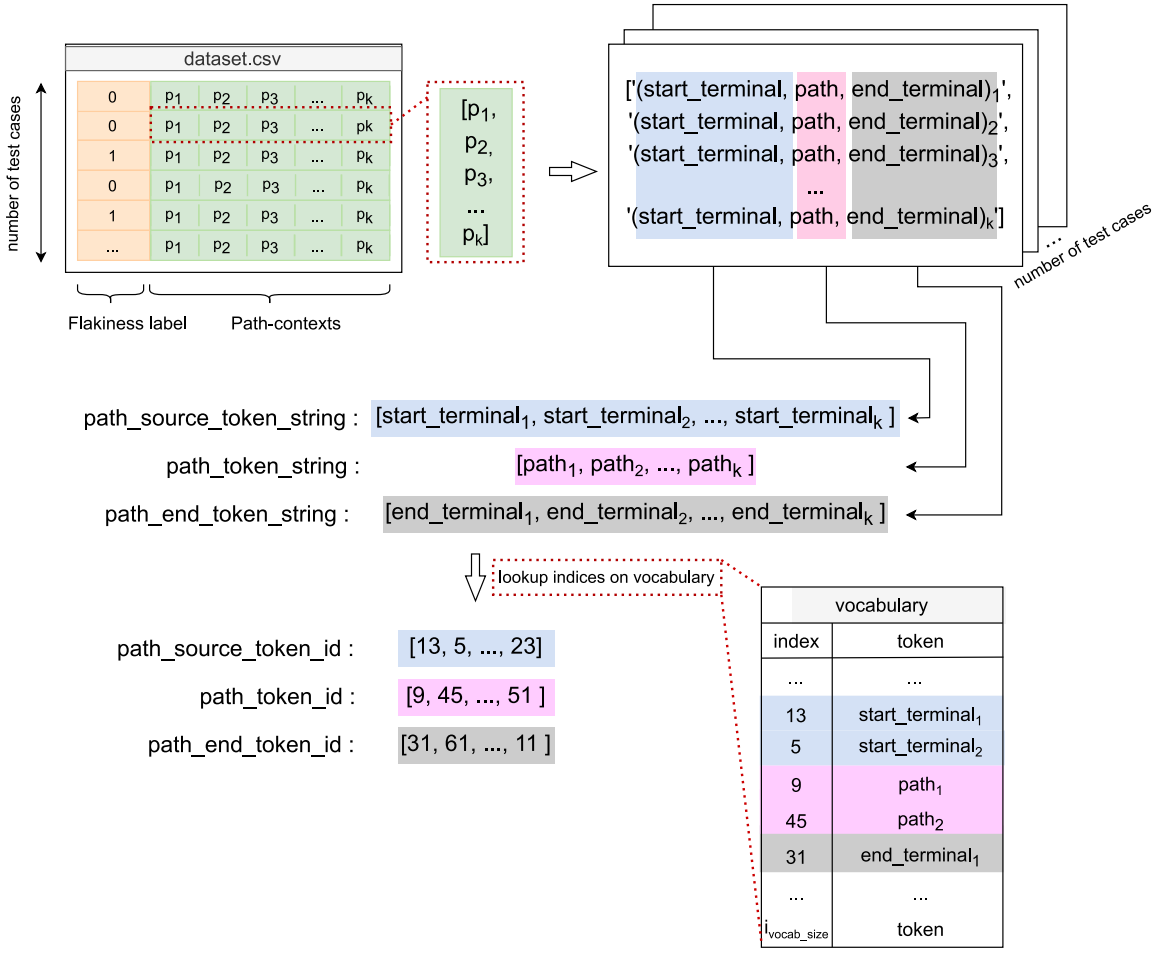


Figure 4.3: Data preprocessing: the dataset rows are read one by one to build the input for the neural network, by categorizing the three parts of path-contexts. For each string of each part we lookup indices on vocabulary. The indices are the input to the model.

4.2.2 *Training Data preprocessing*

In order to feed code representations of test cases along with their flakiness label, we construct one-dimensional tensors out of the training dataset with path-contexts. From each row that represents a test case, we extract the label and the context paths. Path-contexts are sliced into three parts: start terminal string, path string, and end terminal string. We build

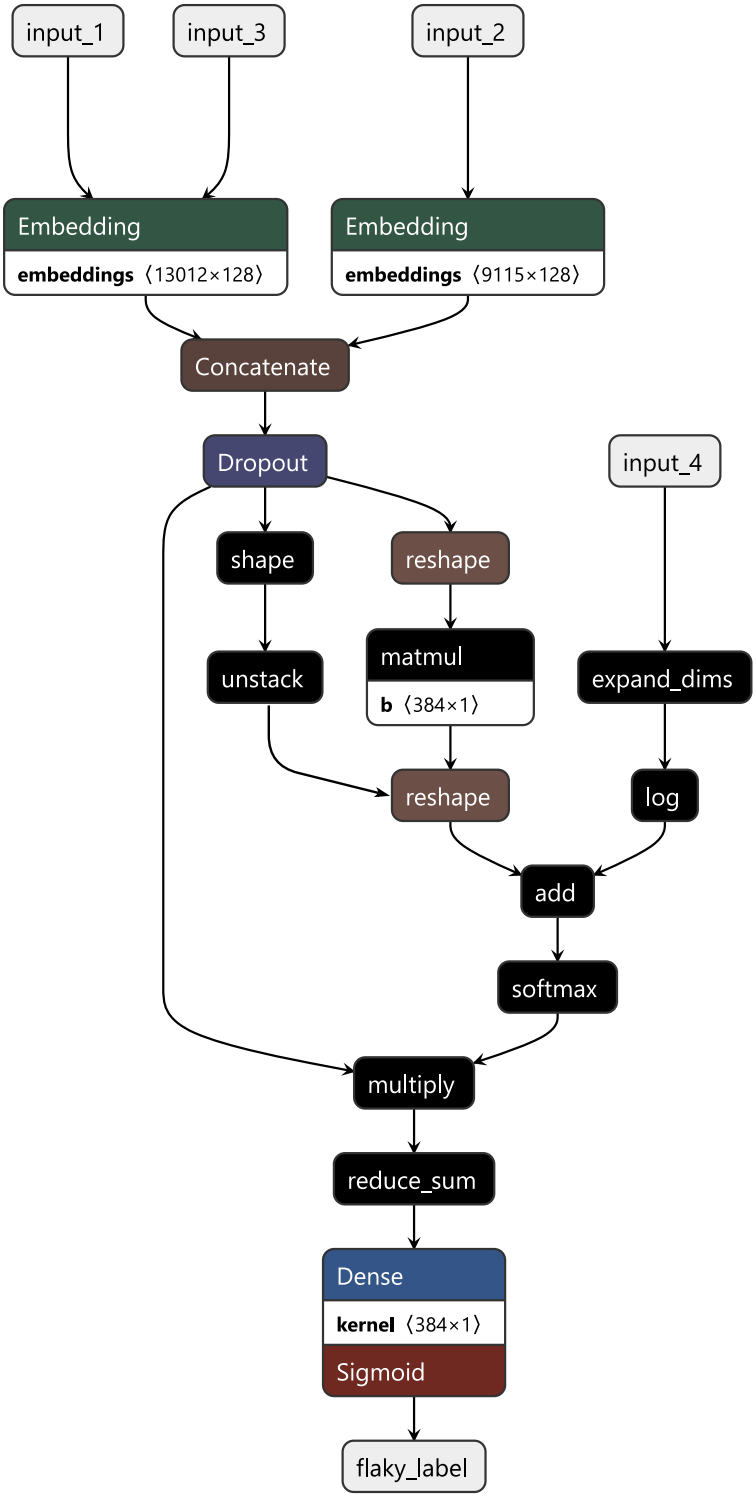


Figure 4.4: Model architecture: a schematic overview of the layers and input data transformations taking place for flaky test classification

tensors for all three parts bounded by the number of path contexts per test case. All three parts of the path-context construct three different vectors. Using the vocabulary look-up tables, we retrieve the indices for every string. This process is visualized in Figure 4.3. At the end of the Training Data preprocessing, we are left with the vectors holding the start terminal string, path string, end terminal string, and vectors with the respective indices. These outcomes are used as inputs for the neural network along with the flakiness label.

4.3 PATH2FLAKE

The flakiness detection is performed by a neural network coupled with an attention mechanism. Neural networks are a fundamental part of many tools and learning tasks in various domains. Learning algorithms such as neural networks have the capability to achieve high performance by learning complex and non-linear relationships [22]. Given this property neural networks make a very desirable option for any learning task. One drawback to be mentioned is the black box nature they have, limiting the interpretability for the classifications they produce [20, 22]. Interpretability is one of the main requirements for Path2Flake. Hence, we leverage attention mechanism and rich code representation, to overcome this issue. We give a description of the model architecture in the following subsections and provide a schematic view in Figure 4.4[2].

4.3.1 *Model Architecture*

The architecture of Path2Flake is inspired by the one used by Alon et al. [8] in code2vec which predicts method names for code snippets. Our neural network that is trained to perform flaky test classification consists of the following layers[3]:

INPUT LAYERS: The input layers are built to accommodate the data input tensors that contain the source, path, and end token indices. They expect vectors with a length equal to the number of *path_contexts_per_test_case*. Also, they expect a mask tensor that asserts which parts of the input row are path-contexts and not padding blank spaces, and is depicted in the Figure 4.4 as input_4. In Figure 4.4, input_1, input_2, and input_3 represent input vectors with indices of start terminal, path, and end terminal token of the path-context, respectively.

EMBEDDING LAYER: To enable network training and transform inputs into numerical machine learning comprehensible inputs, we map the input indices to vector embeddings. The inputs to this layer are the indices of the three parts of each path-context. These indices are used to initialize the embedding matrix which holds the embedding vector for each of the indices in the vocabulary. These vector embeddings are learned along with other parameters of the network during training. The length of the embedding matrix is the same as the vocabulary size. When input indices enter this layer a select function is executed to find and retrieve the embedding of these input indices. This

2 The figure is generated using https://netron.app/

3 An evaluation round is performed to validate a design decision about the layers used in the model. This process is described in Section 5.3.2 and is concerned with measuring the impact of a fully connected layer on the model's performance.

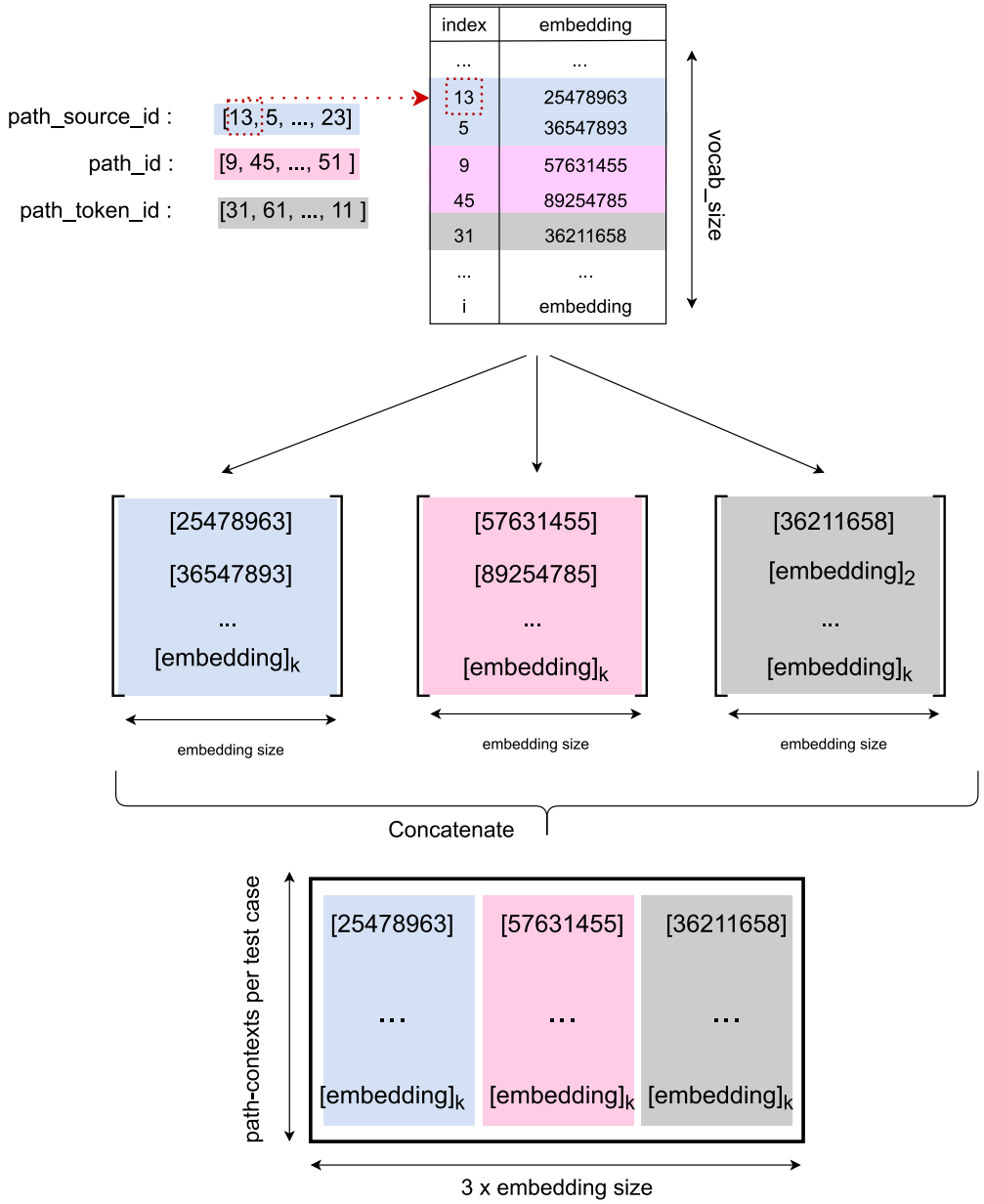


Figure 4.5: Path-contexts embeddings: we initialize the embeddings and map them to the indices of the input token for the source, path, and end part of the path-context. These embeddings are learned during model training. The figure depicts, at a high level, the embedding and concatenation process of the elements for one test case, where k is the number of path-contexts per test case.

process is depicted in Figure 4.5. The output of this layer are three vector embeddings, respectively for each part of path-context; start terminal node, path, and end terminal node.

CONCATENATION LAYER: Up to this point, the test case is represented by three distinct embedding vectors for each path context. However, to perform classification we require a single vector to serve as test case encoding on which the classification is performed. To serve this purpose, we concatenate the three vectors into a single one and push it onto the attention layer to be further merged with the other path-context vectors of the test case.

ATTENTION LAYER: The goal of achieving interpretable outcomes and a unified representation for individual test cases, is supported by the integration of the attention mechanism within Path2Flake. Among two types of attention weights: *soft* and *hard attention*, we use the former one. Soft attention distributes the contribution in calculating the test case code representation among all paths, meaning that all paths are assigned particular attention parameters which are learned during network training. On the contrary, hard attention acts as a 1-hot-encoding, meaning that it focuses on one path context. Soft attention is our choice given that all paths are likely to contribute in discriminating between flaky and non-flaky tests. The path-context embedded vectors are the input of the Attention Layer which multiplies each embedded vector with its respective attention parameter. The length of the vector that holds the attention weights is the same as the number of the path-contexts per test case. As depicted in the Figure 4.4, the embedding vector shape is asserted to be compatible with the parameters of the attention layer for multiplication. The embeddings of each path-context are multiplied with the parameter vector of the same length resulting in attention weight. These parameters are learned during training with the other parameters of the model. Since some of the test cases contain padding we add the mask input to the attention weights. After computing the attention weights, the method applies a *softmax* activation function along the input length axis to obtain a probability distribution over the input sequence for each batch:

$$\text{softmax}(x_i) = \frac{e^{x_i}}{\sum_{j=1}^{n} e^{x_j}}$$

This probability distribution represents the relevance of each path-context element to the output of the layer. By applying the softmax activation function, the attention weights are transformed into a probability distribution that sums to 1 over the input length axis. This ensures that the attention mechanism is able to focus on different parts of the input sequence depending on the task, while also ensuring that the attention weights are normalized and can be interpreted as probabilities. The weights are multiplied with the path-context embeddings to produce a single vector. The attention layer outputs a **single code vector representation** that encodes rich weighted information coming from the AST paths-contexts of the test case. These attention weights provide information to the developer on the flakiness classification outcome. Together with the flakiness label, Path2Flake also outputs the path-contexts with the largest weights that contributed the most to the classification and hence hint at the

root of the flakiness. This information is very important since solely knowing that a test is flaky does not provide actionable information. The main goal in dealing with flaky tests is to tame them as soon as possible, to resume development without the potential of facing intermittent failures.

CLASSIFICATION LAYER: The test case vector representation resulting from the attention layer is the vector holding rich implicit information on the test case semantics and structure encoding patterns of flakiness root causes. The classification is performed by a dense layer that uses the *sigmoid* activation function, which is commonly used in binary classification tasks [22]:

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

This activation function provides an output in the range $(0, 1)$ which can be interpreted as the probability of the predicted output [22]. When executing prediction on individual test cases, the model outputs the flakiness label probability, 5 path-contexts with the highest attention weights.

4.3.2 *Training*

The network is trained through forward passes and backward propagations. A forward pass starts with pushing the preprocessed input to the network along with the label and performing every calculation from the first to the last layer. In the end, the network produces an output value between 0 and 1, which can be interpreted as a confidence probability for the produced output. This output is compared to the ground truth label provided to the network along with the input test case, and the difference is called the loss of the neural network. This loss is then propagated backwards in the network to tweak all the model's parameters in the direction that minimizes this loss. The loss minimization and parameter update are achieved through the optimization algorithm.

Loss function. The loss function of the model is `Binary Cross Entropy (BCE)`, which is suitable for binary classification, as is the flakiness classification. Formally:

$$\mathcal{L}_{BCE} = -\frac{1}{n}\sum_{i=1}^{n} y_i log(p(y_i)) + (1 - y_i) log(1 - p(y_i))$$

where $y_i \epsilon \{0, 1\}$ is the label provided as ground truth and $p(y_i)$ is the predicted probability of the network. This loss function is written for n test cases. The loss for one test case is:

$$\mathcal{L}_{BCE} = -y_i log(p(y)) + (1 - y_i) log(1 - p(y_i))$$

This loss function is minimized by the optimization algorithm.

Optimization. The gradient of the loss with respect to each parameter in the network is calculated by propagating the error back through the network, starting from the output layer to the input layer. To update these network parameters with the aim of minimizing loss, we use the `Adam` optimization algorithm [28]. This algorithm is part of the optimization approaches with adaptive learning rates. The learning rate is a network hyperparameter that controls the step of change applied to the model parameters to reach convergence and

find the minimum of the loss function. Large learning rates lead faster to the minimum but may never reach it due to the big steps it takes. Small learning rates, take smaller steps to reach convergence, but results in a longer training time and are prone to stuck in local minima [14]. Optimization algorithms, such as `Adam`, adapt the learning rate to achieve a better network performance.

The optimization is applied to mini-batches of the training test cases. It significantly improves the performance of the network, since instead of processing the entire training set in one epoch to perform a backpropagation step, the network calculates the gradients for small batches of data. In one epoch there are as many parameter update steps as the number of batches in the dataset. Such practice shortens the convergence time and decreases overfitting.

Regularization. Using regularization approaches in neural network models helps to overcome overfitting [14]. For this purpose, we use `Dropout` on the output of the layer that concatenates three embeddings of the path-context [48]. According to this method, neurons of the neural network are randomly "turned off", (all weights of some neurons are equal to zero) with a probability of p; and are "turned on", with a probability of $q = 1 - p$ during training [22]. During the inference of the model, all neurons are "turned on" and the final activation function is multiplied by q to emulate the behavior of an ensemble of neural networks. The main idea of `Dropout` is to train an ensemble of several neural networks instead of training one neural network, and then average the obtained results [48].

5 EXPERIMENTS

In this chapter, we define the research questions and explain the experiment we conduct to answer them. We present in detail the model selection process and the procedures for training and testing Path2Flake. Furthermore, we elaborate on the dataset preprocessing pipeline.

5.1 OPERATIONALIZATION

We divide our work and approach into measurable components that will be used to evaluate the core claims of this thesis. We build the evaluation framework around the following research questions and define the metrics.

5.1.1 *Research Questions*

> RQ1: How effective are path-context code representations in flaky test classification?

Being able to detect test flaky tests based on path-context embeddings enables us to fulfill such a task in an efficient way. This is particularly important since we conjecture that current solutions rely mostly on costly rerunning techniques. Certainly, our solution can only be useful if the performance of the classifier build following this approach is optimal.

> RQ2: What is the performance of Path2Flake in comparison to state-of-the-art flaky test classifiers?

Having a better understanding, of where Path2Flake stands in terms of performance compared to other tools, aids in identifying the advantages or disadvantages of our work. This comparison helps in identifying the impact our encoding approach and model architecture has on the success of the predictor.

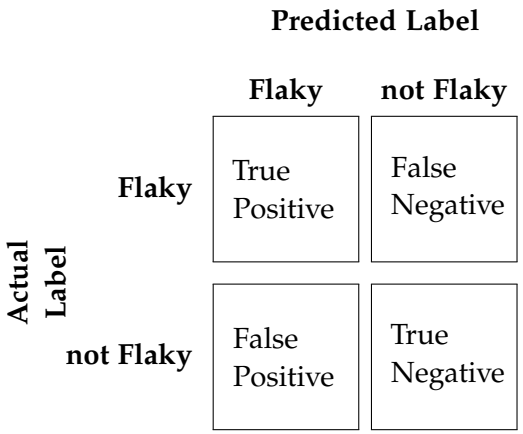
> RQ3: How can Path2Flake deliver outcome explanations to developers using path-context as test case representations?

As opposed to other raw training data, such as text, quantitative measurements, images, and the like; code snippets need a special encoding in order to be used as input data for the machine learning model. The encoded snippets should encapsulate enough information to enable the Path2Flake approach to learn the task of detecting flakiness. Despite the correct classification, Path2Flake aims to deliver explanations for the labels produced. These explanations are based on weighting path-contexts, which serve as pointers inside test cases.

5.1.2 *Evaluation metrics*

In order to quantify the performance of our flakiness classification model, we decide on the evaluation metrics that we employ. These metrics not only gauge the performance of *Path2Flake*, but also provide a benchmark for comparing it with other flakiness classifiers.

Before we present the metrics, we begin by defining the confusion matrix and its components. A confusion matrix provides a summary of the number of correct and incorrect classifications made by the model for each class. The ratios between the correct and incorrect classification for each class provide evidence of the model's performance.



True Positive (TP): The number of flaky tests classified as flaky.
False Positive (FP): The number of non-flaky tests classified as flaky.
True Negative (TN): The number of non-flaky tests classified as non-flaky.
False Negative (FN): The number of flaky tests classified as non-flaky.

These components are combined into the following evaluation metrics that we use to assess the performance of Path2Flake:

- Loss function: measures how close the classification labels produced by Path2Flake are to the ground truth. It guides not only the optimization process, but also serves as a performance indicator. As already mentioned the loss function of our model is BCE.

- Accuracy: measures the performance of the model as the proportion of the correct classifications out of all the classifications made by the model.

$$\text{Accuracy} = \frac{\text{TP} + \text{TN}}{\text{TP} + \text{TN} + \text{FP} + \text{FN}}$$

  This metric is overall useful but it does not account for the correct classification rate for each class individually. This aspect is not to be discarded especially when the dataset on which the model is trained, has a skewed distribution of the number of test cases per class. We need the following complementary metrics.

- Precision: measures how many of the test cases predicted as flaky are actually flaky. A high precision implies a low false positive rate. Such a metric is particularly important in this task because classifying a non-flaky test as flaky poses an unnecessary burden on developers to fix flakiness that is not present.

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}}$$

- Recall: measures how many of the flaky tests in the dataset our classifier was able to detect (also called sensitivity). A high recall rate indicates that the model does not frequently miss flaky tests. This metric is important because classifying a test as non-flaky when it is not such, can encourage developers to ignore it. This can disrupt development and cause intermittent failures.

$$\text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}}$$

- F1-score: is a combination of recall and precision. Overlooking flaky tests can have adverse consequences on the development. However, the time-consuming false positives are not to be discarded either. Since both metrics are important for the task of flakiness classification, F1-score helps in balancing and considering them simultaneously.

$$\text{F1-score} = 2 * \frac{\text{Precision} * \text{Recall}}{\text{Precision} + \text{Recal}}$$

All these metrics are provided by `Keras` framework as optional arguments to `model.compile()` function.

5.2 DATASET

The dataset we use for training and evaluating our classification model is provided by Alshammari et al. [9], who developed FlakeFlagger. This dataset contains 24 projects with flaky and non-flaky test cases written in the Java programming language. In total, there are 21,721 test cases.

After inspecting this dataset, we detected 936 tests that were either empty or did not parse due to syntax errors. Only 5 of these 936 were labeled as flaky. This dataset, which is the ground truth of our training and testing process, was constructed by rerunning the test cases up to 10,000 times by Alshamari et al. [9]. The test suites were also used by authors of other research papers, namely Pinto et al. [44], Fatima et al. [20], and is originally provided by Bell et al. [12]. However, Alshamari et al. relabeled the dataset after rerunning tests using around 5 years of computation time in total. To the best of our knowledge, this is currently the most accurately labeled flaky test dataset.

The authors did not use any approach to increase the probability of detecting flaky tests, that can introduce artificial non-determinism. Hence, this labeled dataset is more representative and closer to a realistic scenario of flaky test occurrence.

It is important to mention, that due to the non-deterministic nature, it is very difficult to reproduce intermittent outcomes of flaky tests. Consequently, this ground truth has some noise that is imposed on the classification model.

To train and test Path2Flake, the data goes through several processes. In Figure 5.1 we provide a high-level visualization of this pipeline.

EXTRACT REPRESENTATIONS: The dataset contains Java test snippets categorized into flaky and non-flaky tests. The first step of processing the dataset is extracting path-contexts from the test cases (step 1 *Extract code representations* in Figure 5.1). To parse the test cases, we override methods presented in `code2vec` by Alon et al. [8]

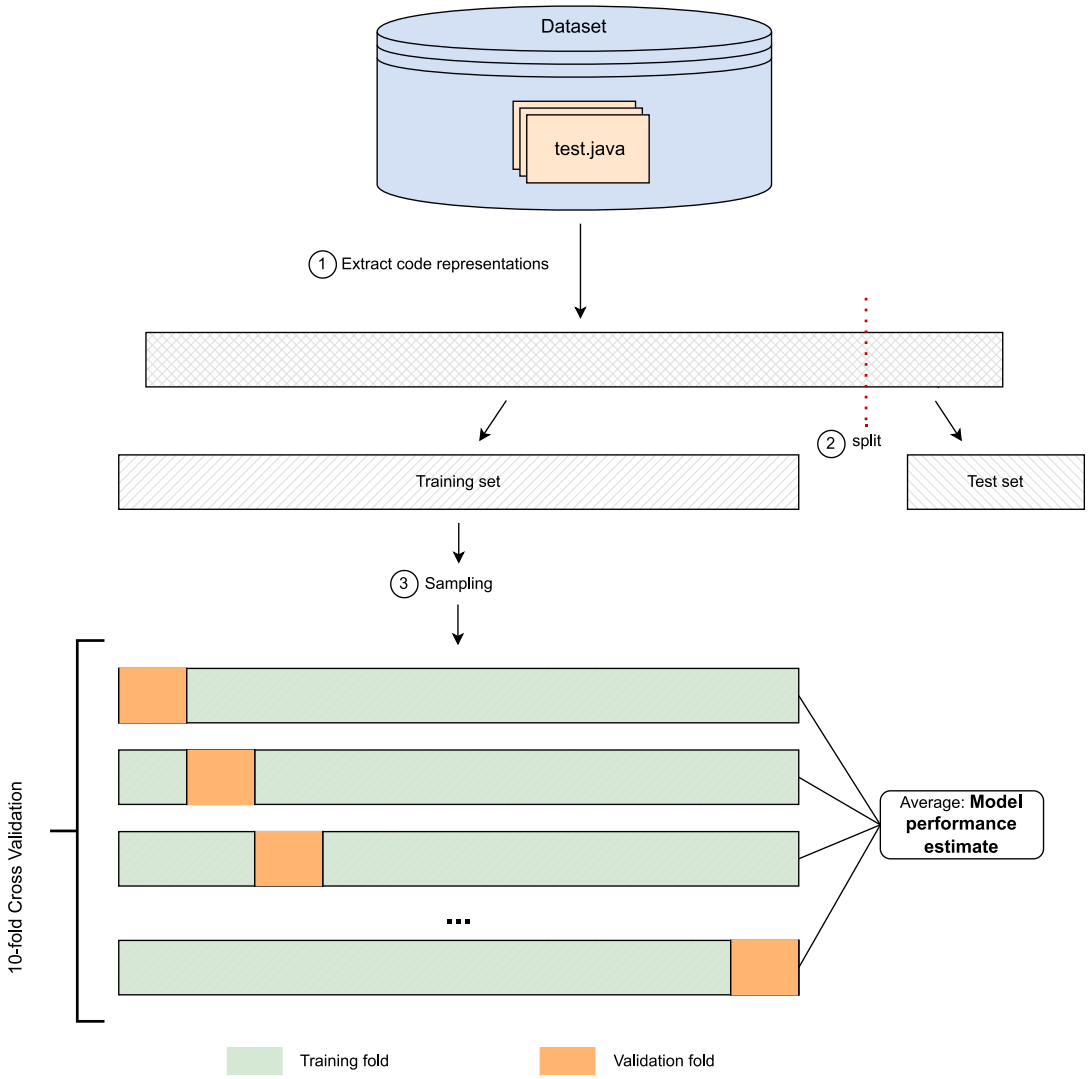


Figure 5.1: Data Pipeline: the various stages that raw data undergoes before being supplied to the neural model. After extracting path-contexts, data is split into a train and test set. Data in the train set is oversampled and further partitioned into 10-folds for the validation process.

and `JavaParser` library [47]. The AST structure allows us to pragmatically work with Java code, specifically, extract syntactic paths between terminal nodes. The extracted syntactic paths along with the label are written into a `csv` file where each row represents a test case. The first element of each row corresponds to the flakiness label and the rest corresponds to n path-contexts.

DATASET SPLIT: After representing test cases as n path-contexts, we randomly split the dataset into a train and test set (step 2 *split* in Figure 5.1). The latter (hold-out-set) will not be used in the training process to remain unseen by the model. This way, after the training when we evaluate the model on the test set, we can obtain a good estimate of the model performance [27]. The split is implemented using the `scikit-learn` Python library [43]. The test set makes up 15% of the dataset, as a commonly used split ratio [22]. The whole dataset has 20,783 test cases which are split into: the hold-out-set with 3,117 test cases(2,991 non-flaky tests, 127 flaky tests), and the training set with 17,666 test cases(16,991 non-flaky tests, 676 flaky tests).

SAMPLING: The large ratio of non-flaky tests to flaky ones indicates that the dataset is highly unbalanced. It contains significantly more non-flaky than flaky tests; specifically, there are 676 flaky tests compared to 16,991 non-flaky ones. Such a ratio has a negative impact on model training, which would end up being highly biased towards the non-flaky class [22].

To balance our training dataset, we leverage sampling approaches [22] (step 3 *Sampling* in Figure 5.1). There are two major categories of sampling techniques that balance the dataset: random oversampling and random undersampling [15]. As the name indicates, random oversampling increases the occurrences of the minority class by randomly selecting and making copies of the existent records in the database. Undersampling does the opposite, by randomly selecting and removing occurrences of the majority class. Both methods achieve a balanced dataset but of different sizes. Since the number of test cases is already not very high, and the minority class is the one we are mostly interested in, we apply oversampling. We implement it using `imblearn` library [32].

ENCODING TEST CASES: As the training set is ready to be processed and fed to the model, we create an input pipeline that consumes data from the `csv` dataset file and prepares it for the network input layer. The implementation is based on `tf.data` API of the `Tensorflow` framework. We fetch data rows from the `csv` file, iterate over the dataset repeatedly, based on the number of epochs, and shuffle the rows to introduce randomness. For each row, we apply the preprocessing function that turns them into the tensor format the network expects. Additionally, we batch the data and filter the vectors that do not fulfill the shape requirements. Each path-context is split into three parts; start terminal vector indices, path vector indices, end terminal vector indices, and path-context mask. These vectors are further used to build embeddings and encode the path-contexts into vectors (Figure 4.5). The path-context embedding is implemented using `Keras` framework built-in layer methods.

5.3 MODEL SELECTION

The performance of Path2Flake is evaluated on the hold-out-set, which consists of test cases unseen to the model during training. As such, this set gives a better estimate of the model's performance and tests its ability to generalize. However, prior to the final evaluation on the hold-out-set, we test the model on validation data and run trials with different configurations of model hyperparameters. The goal of this process is to select the hyperparameter values that yield the best results. These values are calculated during iterations of the validation process. Since this process involves training and testing multiple models, it is also called *model selection* [27].

5.3.1 *Stratified K-Fold Cross Validation*

We perform *Stratified K-Fold Cross Validation* for estimating the performance of Path2Flake and finding the best configuration of hyperparameters. K-Fold Cross Validation splits the training set into k folds, and through k iterations we train and test the model on these folds. This process is visualized in Figure 5.1. Specifically, we train the model on $k-1$ folds of data (marked in green) and test its performance on one fold that was not used during training (marked in orange). With this validation method, we get a better estimate of the loss and other evaluation metrics of the model, by averaging them among k data folds.

Stratified K-Fold indicates that the algorithm splits the dataset preserving its original probability distribution across flaky and non-flaky tests. If we do not use stratified folds, we can end up constructing training and validation folds that do not contain test cases of both classes, limiting the learning ability of the model.

We use Stratified K-Fold Cross Validation when we tune the model hyperparameters and on training after finding the optimal values for the hyperparameters. To implement this validation algorithm we use the `scikit-learn` Python library [43]. The number of folds we use is $k = 10$ as a commonly used value [27].

5.3.2 *HyperParamater Tuning*

The flakiness classification model has several hyperparameters that impact the performance of the model. To identify the optimal configuration of our network, we perform hyperparameter tuning. This process involves training the network on various combinations of hyperparameters and measuring the performance each combination yields.

We use the Hyperband algorithm introduced by Li et al. [33], and implement it using the `keras-tuner` library of `Keras` framework [41]. The hyperparameters that we tune are learning rate, dropout rate, path embedding size, and terminal token embedding size. Hyperband builds all possible configurations of these hyperparameters based on the conditions we predefine on their values:

- path embedding size and terminal token embedding size should be minimally 50 and at most 500. To reduce the overhead we set a stepsize of 78 that reduces the search space.

- learning rate search is bounded by three values (common choices): 0.01, 0.001, and 0.0001.
- dropout rate is bounded by four values (common choices):0.1, 0.15, 0.25, and 0.35.

Hyperparameter tuning is done for each data fold of cross-validation. The reason why we repeat hyperband tuning 10 times on different data splits is that this algorithm's outcome can depend on the initialization of configurations. So, tuning is executed in 10 iterations, to get a better estimate of the best values of hyperparameters.

In each iteration, Hyperband implemented with `keras-tuner` runs trials for evaluating hyperparameter configurations. The tuning process is guided by the objective of minimizing the validation F1-score. We pick this objective because we aim to build a model that yields optimal values for both, precision and recall. At the end of 10-fold cross-validation, we obtain the 10 best hyperparameter configurations and the model's F1-Score. The optimal outcome of this process for each data fold is summarized in Table 5.1.

Table 5.1: Hyperparameter tuning: configurations that yield the best performance of the model in each data split of the 10-fold validation process

Fold	token embedding	path embedding	learning rate	drop rate	Loss	F1-Score
1	128	128	0.001	0.25	0.0595	0.9863
2	284	440	0.001	0.1	0.05324	0.99004
3	440	440	0.0001	0.15	0.0553	0.9860
4	128	440	0.0001	0.1	0.0464	0.9846
5	128	128	0.001	0.1	0.0372	0.9909
6	128	128	0.0001	0.25	0.0688	0.9764
7	206	50	0.001	0.25	0.0807	0.9798
8	50	440	0.001	0.15	0.0648	0.9835
9	128	440	0.0001	0.1	0.0522	0.9835
10	50	50	0.001	0.25	0.0467	0.9880

These configurations are the ones that performed best for the respective data splits. As we can notice on the table, almost every fold results in a different best configuration. This can be a result of the way Hyperband works, which depends on random initialization of the hyperparameter configurations, as mentioned previously.

We visualize the loss and F1-score per epoch of these configurations, in a parallel coordinate graph on Figure 5.2.

We can observe that the hyperparameter values that result in the lowest loss and highest F1-score are: **token embedding** = 128, **path embedding** = 128, **learning rate** = 0.001, **drop rate** = 0.1 Even though this configuration is not the best option for the other folds, the values individually are predominant compared to the other hyperparameter values in other configurations, except for the **path embedding** = 440. However, since the variance of the metrics is not high, and for the sake of the configuration (the metrics are calculated per configuration and not per individual parameter) we will choose the **path embedding** = 128.

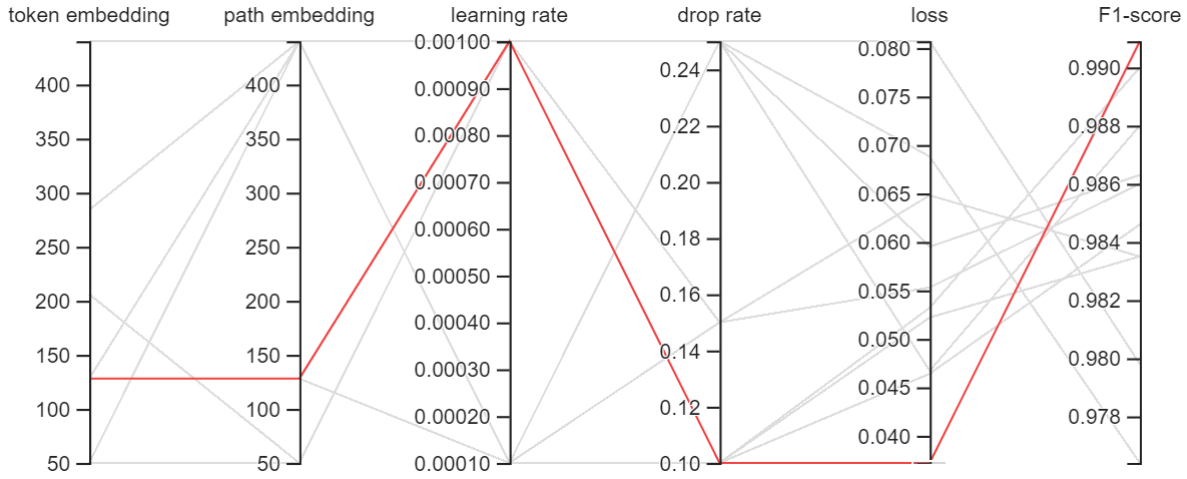


Figure 5.2: Parallel coordinate graph on the hyperparameter configurations: the red line depicts the configuration we chose as the best option among the results. The grey lines represent the other outcome configurations from the hyperparameter tuning process, for each data fold.

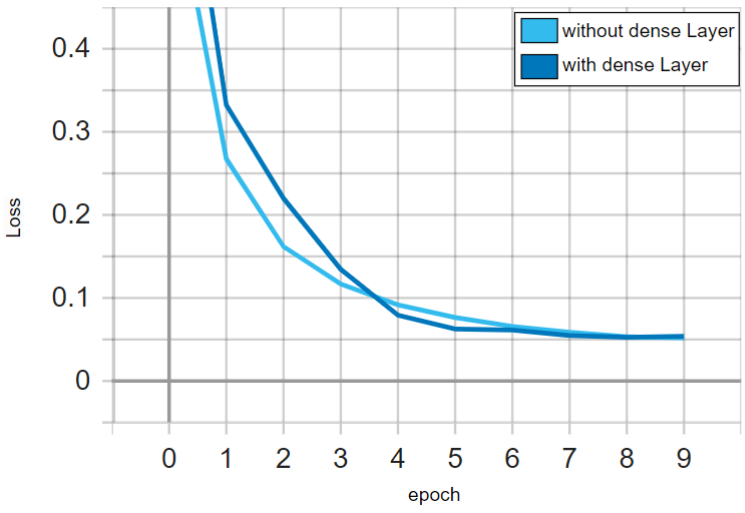


Figure 5.3: Plot of the validation loss through 10 epochs for two models; one with a dense layer after the concatenation layer of embeddings, and another model without such a layer.

Despite, the hyperparameter tuning process we also ran a trial for evaluating the impact of a fully connected layer. This layer is used to further merge the path-context concatenated vector containing three embeddings; the start terminal node, path, and end terminal node. To fulfill this, we apply the*tanh* activation function.

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

This function is a commonly used monotonic nonlinear activation function with output values in the range $(-1, 1)$ and has the potential to increase the expressiveness of the model [22]. For this reason, we run a trial to validate its impact on the model performance.

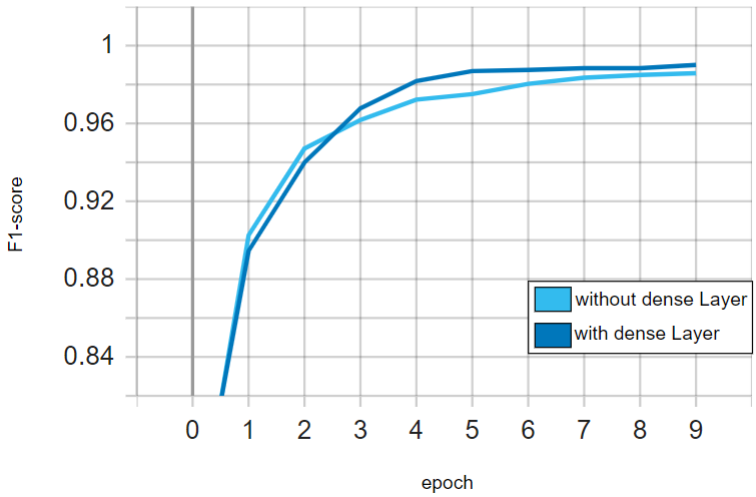


Figure 5.4: Plot of the validation F1-score through 10 epochs for two models; one with a dense layer after the concatenation layer of embeddings, and another model without such a layer.

We train two models for 10 epochs each, where one includes a dense layer while the other does not. In the latter, we bypass the dense layer and directly input the concatenated vector to the attention mechanism. We assess the model's performance using validation sets, and for each epoch, we log and plot the validation loss and validation F1-score using the `TensorBoard` visualization tool provided by `Tensorflow` [1]. In Figure 5.3 we notice that during the first three epochs, the dense layer contributes to a higher loss. However, after the fourth epoch, the model with the dense layer yields better performance. The same pattern is present on the F1-score graph Figure 5.4.

Both models are expected to perform poorly during the first few epochs. The reason why the dense layer model is slightly worse may be due to the higher number of model parameters that must be learned and that require more data to be optimized. However, as the training progresses the missing parameters in the other model, impact the model to learn more complex implicit patterns in data. Thus, it achieves better performance. However, this better performance can be an outcome of overfitting and memorizing the training set. This argument is supported also by the pattern of surpassing the no-dense-layer model only after a few epochs.

All things considered, the difference in performance is very low and insignificant. We follow a widely applicable principle called Occam's razor. It is often used in scientific and philosophical reasoning. In the context of deep learning, it encourages simplicity in model architecture and the use of fewer parameters to prevent overfitting, given that their performance is similar [14, 22]. Hence, we do not include the dense layer in Path2Flake architecture. Removing a fully connected layer also reduces overhead.

5.3.3 *Performance estimation*

With the found hyperparameter configuration, we train the network on the training set with the picked values of that configuration. Before training the model on the full dataset, we try to estimate the evaluation metrics using Stratified K-Fold Cross Validation.

We follow the same steps as in the validation process during hyperparameter tuning, but this time we train the model using `model.fit()` instead of calling `Hyperband` tuner. We log the results and plot the values on graphs using `TensorBoard`. We visualize the model metrics measured on validation sets for each epoch.

The Figure 5.5 shows that the variance of the validation loss is low, with the values varying in a narrow interval. We zoom on the loss for the last epoch, where the mean validation loss is0.05. Such a value is satisfactory for a validation loss. However, we can only get an unbiased metric value on the hold-out-set. Because we use Stratified k-fold validation and oversampling, the calculated loss on that data can be artificially low, affected by such preprocessing. Even though stratified folds try to mimic the distribution of the whole training set, that distribution may not be representative of actual test cases in realistic scenarios.

In Figure 5.6, we show the validation F1-score for each fold. The mean F1-score for the last epoch is 0.98. The same reasoning for the loss results applies in this case.

We trained the models for 10 epochs with a batch size of 128, a typical choice for a dataset with the size of the training set we use. The process of selecting the epoch number and batch size is a trial-and-error process. We did not include them as part of tuning, due to limitations in computation time, and because the first couple of values we tried resulted in satisfactory performance. Selecting a large batch size provides a better estimate of the loss gradient, but is more computationally expensive [22]. On the other hand, a smaller batch size can introduce a regularization effect due to the variance in the estimate of the gradient calculated on this batch [54]. When using Graphics Processing Units (GPUs), it is common for the power of 2 batch sizes (typical power of 2 batch sizes range from 32 to 256) to offer better runtime [22].

5.4 MODEL TRAINING AND TESTING

After we obtain an estimate of the model's performance, we train the final optimal version of the model on the whole training set. We keep the batch size 128 but we double the number of epochs. That is, because the dataset is larger and because there was no deterioration of the metrics as the epoch number increased during validation.

Finally, after training the model, we use the hold-out-set to perform the assessment of the model on unseen data. We evaluate the model's ability to generalize. It is important to

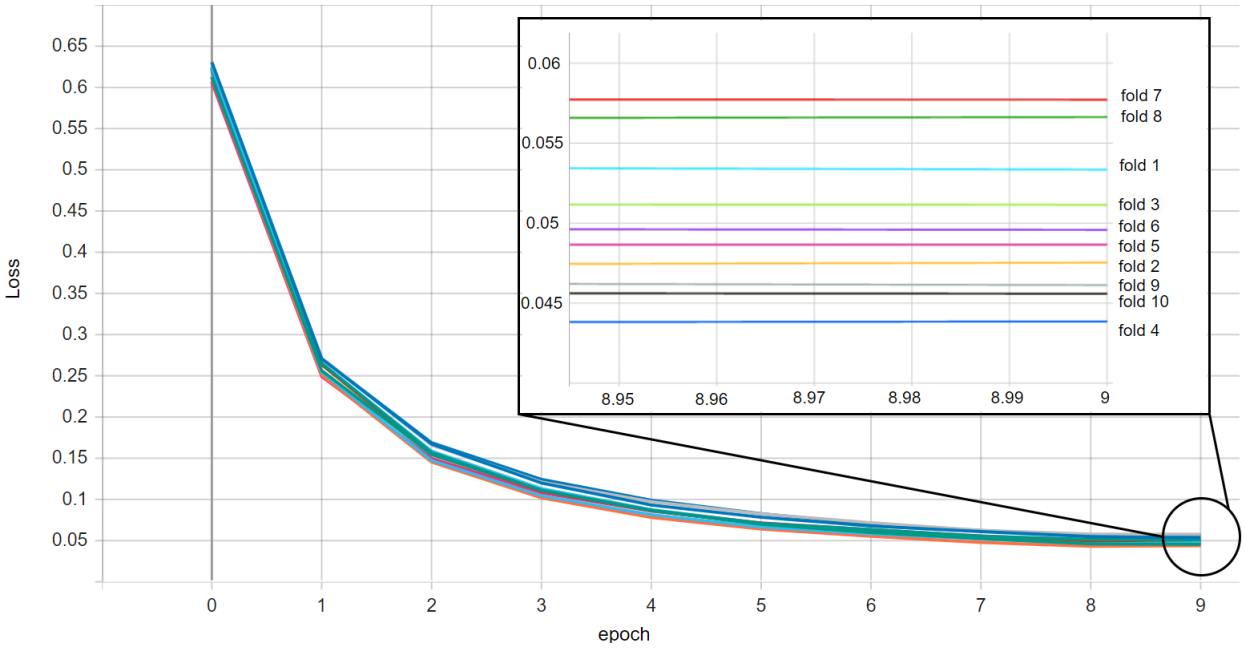


Figure 5.5: Validation loss through 10-fold cross-validation: For each cross-validation iteration Path2Flake is trained for 10 epochs on $k-1$ splits and evaluated in the validation split. Each color represents a data split.

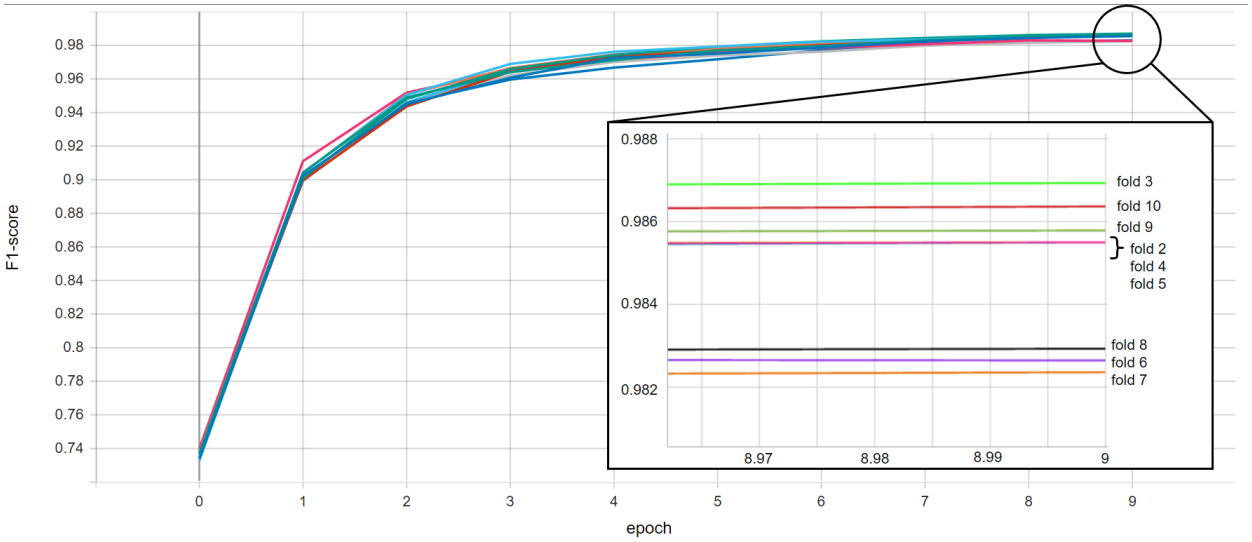


Figure 5.6: Validation F1-score through 10-fold cross-validation: For each cross-validation iteration Path2Flake is trained for 10 epochs on $k-1$ splits and evaluated in the validation split. Each color represents a data split.

emphasize, that this data has not been oversampled and has not interfered in any way with the training and validation process. So, the network has not memorized these test cases on previous passes.

6 EVALUATION

In this chapter, we perform the analysis pertaining to the experiments described in the previous chapter. We present the results on the performance of Path2Flake and answer the research questions. Finally, we discuss the findings and state the possible threats to the validity of the evaluation and results.

6.1 RESULTS

> Answer to RQ1: How effective are path-context code representations in flaky test classification?

We evaluate the performance of our flakiness classifier that learns path-context embeddings and produces flakiness labels for test cases based on these embeddings. The results from the model testing process on the hold-out-set are the following:

Table 6.1: Results on the performance of Path2Flake

Evaluation metric	Value
Loss	0.0898
Accuracy	0.9718
Precision	0.6479
Recall	0.7244
F1-score	0.6840

The neural model is evaluated on a test set with 2,991 non-flaky tests and 127 flaky tests. These results show a loss value that is close to the loss estimated during the 10-fold cross-validation. Whereas, F1-score seems to be 0.296 less than the estimated F1-score. This can be a consequence of the unbalanced test set, that was not sampled in order to be a better representation of the test cases that can occur in realistic scenarios. Even though lower than the estimated F1-score, the results from this assessment show that Path2Flake is able to learn a correspondence between path-contexts and flakiness. Specifically, our model is able to produce a correct label for 97% of its total classifications. However, achieving high accuracy alone does not indicate a good performance of the model. So we analyze the recall and precision values.

Observing the recall and precision values we notice that the model is less likely to miss flaky tests and classify them as non-flaky (Recall = 72.44%) than it is to produce false positives (Precision = 64.79%). Having a higher recall compared to precision, avoids having serious issues such as the manifestation of untamed flaky, but at the cost of wasting more

time on false positives. Flaky tests that are hidden by the classification model can cause intermittent failures that block the development process. Such a scenario can be more adverse than wasting time on a misclassified non-flaky test. However, both aspects are important as already argued in previous sections.

Depending on the use case of classifications, one can decide to give priority to one of these metrics. If Path2Flake is mainly used for facilitating the debugging process, the goal is maximizing the precision. Otherwise, if sensitivity is prioritized, the model should be optimized for recall. Changing the threshold of the classification is a way of prioritizing one of these metrics. If we increase the classification threshold (which currently is 0.5), Path2Flake confidence probability on the flakiness label should be higher in order to label the test as flaky.

One reason for precision being lower than recall is the oversampling of the minority class (flaky tests). Artificially increasing the number of flaky tests in the dataset can introduce bias to the model in classifying this class. Having more flaky tests in the dataset would enhance the model's learning ability of the flaky patterns and enhance its performance.

In order to prove that our model has in fact predictive power and these results are not by chance, we define and obtain results from baseline classifiers. Baseline classifiers are used as a benchmark to compare against; they set a performance threshold that should be surpassed by our model for it to be considered to have predictive power [14, 22]. These baseline model classifications are not informed by the input features of the data but rather perform classification based on naive strategies that account for the class counts of the input data. The strategies that we implemented are:

- stratified: generates random predictions based on the class distribution of the training set.
- uniform: generates random predictions uniformly across both classes.
- most frequent: always predicts the most frequent class in the training set, and if the frequency is the same for both classes on the training set (since we applied oversampling), the frequent class is selected randomly.

We implement these baseline classifiers using the `scikit-learn` Python library, and for each classifier we got the following results:

Table 6.2: Results of baseline classifier using different strategies

Strategy	Accuracy	Precision	Recall	F1-score
stratified	0.4865	0.0335	0.4173	0.0620
uniform	0.5064	0.0433	0.5275	0.0800
most frequent	0.0407	0.0407	1.0	0.0782

As the outcome values indicate, our model yields better performance values than the baseline models for every metric. Aside from the recall of the classifier with the most

frequent strategy, which is expected to have such results as the model assigns the same label for all inputs.

These results prove that Path2Flake has predictive power and is able to discriminate between flaky and non-flaky tests. Using path-contexts as test case representations prove to be informative to the flakiness classification process, which compared to the baseline models shows to have predictive power. Nevertheless, the performance is not outstanding and it needs to be improved given the sensitive task of flaky test classification it should perform.

> Answer to RQ2:What is the performance of Path2Flake in comparison to state-of-the-art flaky test classifiers?

We compare Path2Flake with FlakeFlagger and Flakify, which, to the best of our knowledge, are currently two of the best-performing tools. Both of these tools are trained and evaluated on the same publicly available dataset that we also use. Flakify is a black box classification model that employs CodeBERT, a pre-trained language model, and is fine-tuned to predict flaky tests [20]. FlakeFlagger is a whitebox model that classifies flaky tests based on predefined test smell features, which are extracted after running the test cases [9]. Despite the requirement to run the test case, Flakeflagger also requires access to production code. More information on these tools is provided on Section 3.2.

In Table 6.3 we show the metric values for these tools and our model.

Table 6.3: Results of Path2Flake compared to Flakify and FlakeFlagger

Model	Precision	Recall	F1-score
Path2Flake	64.79%	72.44%	68.40%
FlakeFlagger	60%	72%	65%
Flakify	70%	90%	79%

Compared to FlakeFlagger, our model scores almost 5% higher in precision, 0.44% in recall, and 3.4% higher in F1-score. Not only does Path2Flake not require manual feature engineering, test execution, or access to production code, but it also reduces the test debugging time due to a lower rate of false positives. Even though only slightly, our model is also better at revealing flaky tests (higher recall).

Compared to Flakify, our model is not able to surpass its performance. One reason is the need for more flaky test cases, upon which our model can learn patterns related to flakiness encoded in path-context. Whereas, Flakify is a more sophisticated black box model, which is already trained in a large language dataset. This facilitates the tuning towards the flaky test prediction, hence yielding a better performance for the same dataset. However, being a black box model Flakify is able to deliver only a label and no interpretability on its prediction. The lack of such a requirement provides flexibility in building a complex model architecture that is optimized for producing correct predictions. Also, Flakify uses an AST based technique for statically detecting and only retaining statements that match eight

test smells (features used by FlakeFlagger) in the test code when the length of the test case exceeds the predefined limit.

> Answer to RQ3: How can Path2Flake deliver outcome explanations to developers using path-context as test case representations?

One of the objectives of Path2Flake is to provide hints at flakiness root causes along with the classification label. Our neural model incorporates an attention layer that serves this purpose. Path-context are pointers to test case locations. The attention layer weights these path-contexts based on their importance in producing the output.

The output of the classifier when a test case is provided as input is the following:

```
[0.6988]
0.03361 context: l,(LongLiteralExpr2)^(MethodCallExpr)_(NameExpr3),waitjobs

0.02986 context: testAsyncTask,(NameExpr2)^(MethodDeclaration)_(BlockStmt)_(
    ExpressionStmt)_(MethodCallExpr0)_(IntegerLiteralExpr1),0

0.02308 context: void,(VoidType1)^(MethodDeclaration)_(BlockStmt)_(ExpressionStmt)_(
    MethodCallExpr0)_(NameExpr3),assertequals

0.02216 context: void, (VoidType1)^(MethodDeclaration)_(BlockStmt)_(ExpressionStmt)_(
    MethodCallExpr0)_(MethodCallExpr2)_(NameExpr2),count

0.02215 context: void,(VoidType1)^(MethodDeclaration)_(BlockStmt)_(ExpressionStmt)_(
    MethodCallExpr0)_(NameExpr3),waitjobs
```

This outcome is produced for this flaky test case:

```
@Deployment public void testAsyncTask(){
  waitJobs(5000L,200L);
 assertEquals(0,managementService.createJobQuery().count());
}
```

The first number indicates the probability of the test being flaky and the rest of the output lists the top 5 context-paths with the largest attention weight. In the path-contexts ^ and _ symbolize ↑ and ↓ direction, respectively. The percentages of the top 5 most important path-contexts are almost the same, namely around 2%. The number of the path-context that we use to represent a test case is 50, which means the model is distributing the attention weights almost uniformly for all the path-contexts. This implies that the model is not able to learn a weight for path-contexts that mostly contribute to flakiness and hint at root causes. One reason is the limited number of flaky tests in the dataset, on which the model can learn path-context patterns related to flakiness root causes. Also, the test cases in the dataset, belong to 24 different projects. The test patterns among these projects can vary, and since the number of flaky test cases per project is low, the model finds it difficult to learn and generalize across projects.

Furthermore, the parameters related to path-context extraction need some tuning and trials to reveal better values for them. These parameters are path length, path width, and the number of path contexts per test case, and they directly impact the amount of information encoded in test case representations.

We are optimistic about the potential that these representations have, given the performance they yield in detecting flaky tests. For this reason, we believe that the improvement in parameters related to path-context size and a larger dataset of flaky tests will enhance the performance of the model in pointing at flakiness root causes.

6.2 DISCUSSION

We built a model that learns embeddings for path-context representations of test cases and uses them to detect flakiness. The results prove the effectiveness of path-context embeddings as test case representations in flaky test classification. The prediction power that Path2Flake has, is evident from the comparison with the baseline classifiers which do not learn from test cases to perform prediction, but rather employ naive random strategies.

Compared to state-of-the-art, our model was able to perform better than one of them. Additionally, even though Flakify scored higher, our model follows an approach that better serves the developers' needs. In terms of efficiency, Path2Flake is advantageous, since it does not require manual feature engineering, test executions, or production code access. The architecture of Path2Flake and the meaningful path-context representations provide the infrastructure to deliver hints at flakiness's root causes. The attention layer weights do not discriminate between paths-contexts that hint at flakes and those that do not. However, recognizing the factors that contributed to these outcomes helps us address them and improve the model's ability to learn this task. One of the improvements we can introduce is experimenting with the path width and path length of the test case representations. Additionally, since limiting the number of path-contexts per test case results in discarding some of them, we can improve the means by which we do that. We can inform the path-sampling process to retain path-contexts whose terminal tokens are part of a vocabulary that is frequent among flaky tests, or common among the test smell patterns. This vocabulary can be obtained from the study done by Pinto et al [44].

The model's learning ability is limited by the low number of flaky test cases in the dataset. The bias introduced by the oversampling of the minority class is reflected in the evaluation metrics and the attention weights that cannot discriminate between path contexts that hint at flakiness root causes and those that do not. The fact that the dataset incorporates test cases from 24 projects, can imply a high variability in flaky test patterns and each pattern being represented only by a few test cases. Hence, the model faces difficulties in learning these patterns.

This task is difficult and we are aware of the misclassification's adverse implications on development. Even though our classifier surpassed FlakeFlagger in performance, leaving 27% of flaky tests undetected leads to test failures and lower reliability on our classifier. The false positives also will waste developers' time in debugging misclassified non-flaky tests.

Despite the limitations, the evaluation proves Path2Flake to be useful and valid. By addressing the limitation and further refining the process of extracting the code representation,

we believe the model performance will improve. This work is the first attempt at this approach, which seems promising for further exploration.

6.3 THREATS TO VALIDITY

Our approach might face challenges to its validity, with some arising from the implementation and others stemming from the characteristics of the data employed.

6.3.1 *Internal Validity*

Our approach and methodology can have confounds that might compromise the results drawn from the experiments. The dataset on which Path2Flake is trained and tested is oversampled due to uneven distribution of the number of flaky and non-flaky tests. To balance the dataset, the number of flaky tests is artificially increased by random duplication, and such duplication is significant due to the high gap between the flaky and non-flaky tests. This can cause the model to become too sensitive to the specific characteristics of these flaky tests, leading to overfitting. To mitigate this threat, we follow several practices that can reduce overfitting. We use the dropout regularization technique and apply the Adam optimization algorithm on mini-batches of the dataset.

The implementation of Path2Flake may introduce additional threats to the validity through potential bugs or errors. To increase our confidence in our implementation, we used well-regarded libraries, frameworks, and visualization tools such as: *scikit-learn* [43] and *imblearn* [32] libraries, *Keras* [41] and *Tensorflow* [1] frameworks, *Tensorboard*[1] and *Netron*[2] visualization tools, etc.

6.3.2 *External Validity*

External validity threats compromise the ability of the model to generalize to other test cases outside our dataset. The dataset that we use contains test cases written in Java from 24 projects. As such, this dataset may not represent all flaky tests in other projects and written in other programming languages. This might limit the generalizability of our results. We believe that the nature of flakiness patterns is not significantly different in other programming languages. Our approach facilitates this investigation since it is applicable to test cases in other programming languages if we use parses for that particular language to build ASTs. Another external threat is the ground truth of the dataset. It is challenging to confidently label flaky tests as such due to their non-deterministic nature. Provably identifying all the flaky tests within a project remains an elusive goal. In an effort to minimize the impact of this threat, we employed a dataset that was annotated after executing the test cases up to 10,000 times. To the best of our knowledge, this is currently the most accurately labeled, publicly available dataset of flaky tests.

1 https://tensorboard.dev/
2 https://netron.app/

7 CONCLUDING REMARKS

7.1 CONCLUSION

The prevalence of flaky tests and their adverse consequences on development motivated our work in finding a preventive solution that detects flaky tests and provides insights into where the flakiness stems from. We developed *Path2Flake*, a static approach that uses deep learning and leverages code representations to effectively learn and perform the task of classifying flaky tests. We use AST path-contexts as test representations coupled with model architecture to provide interpretability of the classification outcomes.

Our results demonstrate that representing test cases as path-contexts enable the Path2Flake model to produce rich encodings and discriminate between flaky and non-flaky tests. The evaluation outcomes obtained from the model testing compared to baseline models' performance and state-of-the-art provide evidence for the effectiveness of our test case representation in flakiness detection. There are some limitations posed to Path2Flake that are concerned with the unbalanced dataset on which the model is trained, and with the internal confounds. The evaluation results do not provide solid evidence for the ability of the model to precisely point to flakiness root causes. However, Path2Flake as the first attempt at this approach, provides the infrastructure to classify flaky tests and give explanations for the classification. Path-contexts are meaningful test representations and can be useful pointers into the test snippet. The performance of Path2Flake proves it to be promising and worthy of further investigation.

7.2 FUTURE WORK

Given that evaluation results provide evidence for our approach to be promising for further exploration, there are several perspectives that can contribute to better performance. Our approach of representing test snippets with path-contexts imposes some limitations. The path width, path length, and the number of path-contexts that will represent one test case are predefined. To maximize the relevant information of test flakiness encoded in these path-contexts, we should experiment and evaluate more configurations of these variables. Ideally, these parameters should be adaptable to the length of the test cases.

Let us consider the predefined value of the number of path-contexts that are retained to represent a test case. Some of the test cases after parsing produce more path-contexts than the preset condition suggests. We perform path sampling which discards path-contexts at random to fulfill the condition. Among these discarded path-contexts can be the ones that are highly influential in the flakiness classification. One possible solution is to inform the sampling process by prioritizing path-contexts whose terminal tokens are part of a flaky test vocabulary. Since these path-contexts can help to better discriminate between flaky and non-flaky tests they should be kept in the representation pool of the test case. As already described, Pinto et al. have conducted research on the correspondence between flakiness

and syntactical patterns, whose outcome is a vocabulary with tokens that frequently occur among flaky tests [44]. We can use this vocabulary as a reference for the procedure.

To further enhance our approach, we can consider splitting the path string of the path-contexts into its constituent building components, which correspond to the non-terminal nodes. This will introduce flexibility in representing tests with a richer vocabulary of paths. This step goes further into fighting the sparsity. Such an improvement can increase the classifier's ability to generalize to unseen test cases.

Another direction this research can take is remaining on the AST based representations but leaning more toward the flakiness pattern detection using convolution operation. Mou et al. [39] use TBCNN to classify code snippets according to their functionality, by detecting frequently occurring patterns among these functionalities. Leveraging TBCNN in the task of detecting flakiness can be advantageous as it better captures tree structure and code patterns. It also overcomes the constraint of the varying length of the test snippets by using the continuous binary tree.

In conclusion, there are several approaches worth pursuing. The findings and contributions of this thesis provide a solid foundation for future research.

BIBLIOGRAPHY

[1] Martín Abadi et al. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Software available from tensorflow.org. 2015. URL: https://www.tensorflow.org/.

[2] Miltiadis Allamanis, Earl T. Barr, Premkumar Devanbu, and Charles Sutton. "A Survey of Machine Learning for Big Code and Naturalness." In: 51.4 (2018).

[3] Miltiadis Allamanis, Hao Peng, and Charles Sutton. "A Convolutional Attention Network for Extreme Summarization of Source Code." In: (2016).

[4] Miltiadis Allamanis and Charles Sutton. "Mining Idioms from Source Code." In: *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. FSE 2014. Hong Kong, China: Association for Computing Machinery, 2014, 472–483. ISBN: 9781450330565.

[5] Miltiadis Allamanis, Daniel Tarlow, Andrew D. Gordon, and Yi Wei. "Bimodal Modelling of Source Code and Natural Language." In: *Proceedings of the 32nd International Conference on International Conference on Machine Learning - Volume 37*. ICML'15. Lille, France: JMLR.org, 2015, 2123–2132.

[6] Uri Alon, Shaked Brody, Omer Levy, and Eran Yahav. "code2seq: Generating Sequences from Structured Representations of Code." In: (2018).

[7] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. "A General Path-Based Representation for Predicting Program Properties." In: *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI 2018. Philadelphia, PA, USA: Association for Computing Machinery, 2018, 404–419. ISBN: 9781450356985.

[8] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. "Code2Vec: Learning Distributed Representations of Code." In: *Proc. ACM Program. Lang.* 3.POPL (Jan. 2019), 40:1–40:29. ISSN: 2475-1421.

[9] Abdulrahman Alshammari, Christopher Morris, Michael C Hilton, and Jonathan Bell. "FlakeFlagger: Predicting Flakiness Without Rerunning Tests." In: *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)* (2021), pp. 1572–1584.

[10] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. "Neural machine translation by jointly learning to align and translate." English (US). In: *arXiv* (2014).

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

[12] Jonathan Bell, Owolabi Legunsen, Michael Hilton, Lamyaa Eloussi, Tifany Yung, and Darko Marinov. "DeFlaker: Automatically Detecting Flaky Tests." In: *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. 2018, pp. 433–444.

[13] Steven Bird, Ewan Klein, and Edward Loper. *Natural language processing with Python: analyzing text with the natural language toolkit*. " O'Reilly Media, Inc.", 2009.

[14] Christopher M. Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Berlin, Heidelberg: Springer-Verlag, 2006. ISBN: 0387310738.

[15] Paula Branco, Luís Torgo, and Rita P. Ribeiro. "A Survey of Predictive Modeling on Imbalanced Domains." In: *ACM Comput. Surv.* 49.2 (2016).

[16] Bruno Camara, Marco Silva, Andre Endo, and Silvia Vergilio. "On the Use of Test Smells for Prediction of Flaky Tests." In: SAST '21. Joinville, Brazil: Association for Computing Machinery, 2021, 46–54. ISBN: 9781450385039.

[17] François Chollet et al. *Keras*. https://keras.io. 2015.

[18] Arie Van Deursen, Leon Moonen, Alex Bergh, and Gerard Kok. "Refactoring Test Code." In: *Proceedings of the 2nd International Conference on Extreme Programming and Flexible Processes in Software Engineering (XP2001*. 2001, pp. 92–95.

[19] Moritz Eck, Fabio Palomba, Marco Castelluccio, and Alberto Bacchelli. "Understanding flaky tests: the developer's perspective." In: *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, 2019.

[20] Sakina Fatima, Taher A. Ghaleb, and Lionel Briand. "Flakify: A Black-Box, Language Model-Based Predictor for Flaky Tests." In: *IEEE Transactions on Software Engineering* (2022), pp. 1–17.

[21] Martin Fowler. "Eradicating non-determinism in tests." In: (). URL: https://martinfowler.com/articles/nonDeterminism.html.

[22] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. http://www.deeplearningbook.org. MIT Press, 2016.

[23] Alex Gyori, Ben Lambeth, August Shi, Owolabi Legunsen, and Darko Marinov. "NonDex: A Tool for Detecting and Debugging Wrong Assumptions on Java API Specifications." In: FSE 2016. Seattle, WA, USA: Association for Computing Machinery, 2016, 993–997. ISBN: 9781450342186.

[24] Vincent J. Hellendoorn and Premkumar Devanbu. "Are Deep Neural Networks the Best Choice for Modeling Source Code?" In: *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ESEC/FSE 2017. Paderborn, Germany: Association for Computing Machinery, 2017, 763–773. ISBN: 9781450351058.

[25] Abram Hindle, Earl T. Barr, Zhendong Su, Mark Gabel, and Premkumar Devanbu. "On the Naturalness of Software." In: *Proceedings of the 34th International Conference on Software Engineering*. ICSE '12. Zurich, Switzerland: IEEE Press, 2012, 837–847. ISBN: 9781467310673.

[26] Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. "Summarizing Source Code using a Neural Attention Model." In: *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Berlin, Germany: Association for Computational Linguistics, Aug. 2016, pp. 2073–2083.

[27] Gareth James, Daniela Witten, Trevor Hastie, and Robert Tibshirani. In: *An introduction to statistical learning*. New York, NY, USA: Springer, 2013. ISBN: 9781461471370.

[28] Diederik P. Kingma and Jimmy Ba. "Adam: A Method for Stochastic Optimization." In: *CoRR* abs/1412.6980 (2014).

[29] Wing Lam, Patrice Godefroid, Suman Nath, Anirudh Santhiar, and Suresh Thummalapenta. "Root Causing Flaky Tests in a Large-Scale Industrial Setting." In: ISSTA 2019. Beijing, China: Association for Computing Machinery, 2019, 101–111. ISBN: 9781450362245.

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

[31] Johannes Lampel, Sascha Just, Sven Apel, and Andreas Zeller. "When Life Gives You Oranges: Detecting and Diagnosing Intermittent Job Failures at Mozilla." In: *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ESEC/FSE 2021. Athens, Greece: Association for Computing Machinery, 2021, 1381–1392. ISBN: 9781450385626.

[32] Guillaume Lemaître, Fernando Nogueira, and Christos K. Aridas. "Imbalanced-learn: A Python Toolbox to Tackle the Curse of Imbalanced Datasets in Machine Learning." In: *Journal of Machine Learning Research* 18.17 (2017), pp. 1–5. URL: http://jmlr.org/papers/v18/16-365.

[33] Lisha Li, Kevin Jamieson, Giulia DeSalvo, Afshin Rostamizadeh, and Ameet Talwalkar. *Hyperband: A Novel Bandit-Based Approach to Hyperparameter Optimization*. 2018. arXiv: 1603.06560 [cs.LG].

[34] Octavio Loyola-González. "Black-Box vs. White-Box: Understanding Their Advantages and Weaknesses From a Practical Point of View." In: *IEEE Access* 7 (2019), pp. 154096–154113.

[35] Qingzhou Luo, Farah Hariri, Lamyaa Eloussi, and Darko Marinov. "An Empirical Analysis of Flaky Tests." In: FSE 2014. Hong Kong, China: Association for Computing Machinery, 2014. ISBN: 9781450330565.

[36] Thang Luong, Hieu Pham, and Christopher D. Manning. "Effective Approaches to Attention-based Neural Machine Translation." In: *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*. Lisbon, Portugal: Association for Computational Linguistics, Sept. 2015, pp. 1412–1421.

[37] "Manage flaky tests." In: (2023). URL: https://learn.microsoft.com/en-us/azure/devops/pipelines/test/flaky-test-management?view=azure-devops#tests-marked-as-flaky.

[38] John Micco. "Flaky tests at Google and how we mitigate them." In: ACM, 2016. URL: https://testing.googleblog.com/2016/05/flaky-tests-at-google-and-how-we.html.

[39] Lili Mou, Ge Li, Lu Zhang, Tao Wang, and Zhi Jin. "Convolutional Neural Networks over Tree Structures for Programming Language Processing." In: *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*. AAAI'16. Phoenix, Arizona: AAAI Press, 2016, 1287–1293.

[40] Andrew Ng, Kian Katanforoosh, and Younes Bensouda Mourri. *Improving Deep Neural Networks: Hyperparameter Tuning, Regularization and Optimization*. URL: https://www.coursera.org/lecture/deep-neural-network/adam-optimization-algorithm-w9VCZ.

[41] Tom O'Malley, Elie Bursztein, James Long, François Chollet, Haifeng Jin, Luca Invernizzi, et al. *Keras Tuner*. https://github.com/keras-team/keras-tuner. 2019.

[42] Fabio Palomba and Andy Zaidman. "Does Refactoring of Test Smells Induce Fixing Flaky Tests?" In: *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)* (2017), pp. 1–12.

[43] F. Pedregosa et al. "Scikit-learn: Machine Learning in Python." In: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830.

[44] Gustavo Pinto, Breno Miranda, Supun Dissanayake, Marcelo d'Amorim, Christoph Treude, and Antonia Bertolino. "What is the Vocabulary of Flaky Tests?" In: New York, NY, USA: Association for Computing Machinery, 2020. ISBN: 9781450375177.

[45] Per Runeson, Emelie Engström, and Margaret-Anne Storey. "The Design Science Paradigm as a Frame for Empirical Software Engineering." In: *Contemporary Empirical Methods in Software Engineering*. Ed. by Michael Felderer and Guilherme Horta Travassos. Springer International Publishing, 2020, pp. 127–147.

[46] Mojtaba Shahin, Muhammad Ali Babar, and Liming Zhu. "Continuous Integration, Delivery and Deployment: A Systematic Review on Approaches, Tools, Challenges and Practices." In: *IEEE Access* 5 (2017), pp. 3909–3943.

[47] Nicholas Smith, Danny van Bruggen, and Federico Tomassetti. *Java Parser: Visited*. URL: https://javaparser.org/.

[48] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. "Dropout: A Simple Way to Prevent Neural Networks from Overfitting." In: *J. Mach. Learn. Res.* 15.1 (2014), 1929–1958.

[49] Tom Taulli. *Testing in the Digital Age: AI Makes the Difference*. New York, NY: Apress, 2018.

[50] Valerio Terragni, Pasquale Salza, and Filomena Ferrucci. "A Container-Based Infrastructure for Fuzzy-Driven Root Causing of Flaky Tests." In: *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: New Ideas and Emerging Results*. ICSE-NIER '20. Seoul, South Korea: Association for Computing Machinery, 2020, 69–72. ISBN: 9781450371261.

[51] Michele Tufano, Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, Andrea De Lucia, and Denys Poshyvanyk. "An Empirical Investigation into the Nature of Test Smells." In: ASE 2016. Singapore, Singapore: Association for Computing Machinery, 2016, 4–15. ISBN: 9781450338455.

[52] Dheeraj Vagavolu, Karthik Chandra Swarna, and Sridhar Chimalakonda. "A Mocktail of Source Code Representations." In: *CoRR* abs/2106.10918 (2021). arXiv: 2106.10918.

[53] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. "Attention is All you Need." In: *Advances in Neural Information Processing Systems*. Ed. by I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett. Vol. 30. Curran Associates, Inc., 2017.

[54] D. Wilson and Tony Martinez. "The general inefficiency of batch training for gradient descent learning." In: *Neural networks : the official journal of the International Neural Network Society* 16 (Jan. 2004), pp. 1429–51.

[55] W.E. Wong, J.R. Horgan, S. London, and H. Agrawal. "A study of effective regression testing in practice." In: *Proceedings The Eighth International Symposium on Software Reliability Engineering*. 1997, pp. 264–274.

[56] Fabian Yamaguchi, Nico Golde, Daniel Arp, and Konrad Rieck. "Modeling and Discovering Vulnerabilities with Code Property Graphs." In: *2014 IEEE Symposium on Security and Privacy*. 2014, pp. 590–604.

[57] Kechi Zhang, Wenhan Wang, Huangzhao Zhang, Ge Li, and Zhi Jin. "Learning to Represent Programs with Heterogeneous Graphs." In: ICPC '22. Virtual Event: Association for Computing Machinery, 2022, 378–389. ISBN: 9781450392983.

[58] Yin Zhang, Rong Jin, and Zhi-Hua Zhou. "Understanding bag-of-words model: A statistical framework." In: *International Journal of Machine Learning and Cybernetics* 1 (Dec. 2010), pp. 43–52.