2022-09

Application of Transformers in Software Test Case Prioritization

Jabbar, Emad

UNIVERSITY OF CALGARY

Application of Transformers in Software Test Case Prioritization

by

Emad Jabbar

A THESIS

SUBMITTED TO THE FACULTY OF GRADUATE STUDIES

IN PARTIAL FULFILMENT OF THE REQUIREMENTS FOR THE

DEGREE OF MASTER OF SCIENCE

GRADUATE PROGRAM IN ELECTRICAL ENGINEERING

CALGARY, ALBERTA

SEPTEMBER, 2022

Abstract

Most automated software testing tasks can benefit from the abstract representation of test cases. Traditionally, this is done by encoding test cases based on their code coverage. Specification-level criteria can replace code coverage to better represent test cases' behavior, but they are often not cost-effective. In this paper, we hypothesize that execution traces of the test cases can be a good alternative to abstract their behavior for automated testing tasks. We propose a transformer-based embedding approach, Transformer Test2Vec, that maps test execution traces to a latent space. We evaluate this representation in the test case prioritization (TP) task. Our default TP method is based on the similarity of the embedded vectors to historical failing test vectors. We also study an alternative based on the diversity of test vectors. Finally, we propose a method to decide which TP to choose, for a given test suite. The experiment is based on several real and seeded faults with over a million execution traces. Results show that our proposed TP improves the best embedding alternative by 40.62% in terms of the median normalized rank of the first failing test case (FFR). It outperforms traditional code coverage-based approaches by 20.72% and 72.59% in terms of median APFD and median normalized FFR.

Preface

A journal paper based on this work has been submitted to ACM Transactions on Software Engineering and Methodology (TOSEM):

- Emad Jabbar, Soheila Zangeneh, Hadi Hemmati, and Robert Feldt. Test2vec: An execution trace embedding for test case prioritization. Submitted to ACM Transactions on Software Engineering and Methodology (TOSEM) 2022. arXiv preprint arXiv:2206.15428

I was the lead investigator, responsible for all major areas of concept formation, data collection and analysis, as well as the majority of manuscript composition. S. Zanganeh was involved in the early stages of concept formation. H. Hemmati and R. Feldt were the supervisory author on this project and was involved throughout the project in concept formation and manuscript edits.

Acknowledgements

I would like to appreciate the kind support from my graduate supervisor, Dr. Hemmati. During this project, I would like to express my gratitude to him for his encouragement and advice. The meticulous editing he provided was instrumental in producing this thesis, and if it had not been for him, the roadblocks would not have been able to be overcome.

In addition, I could not have accomplished so much without the support, encouragement, and care of my family and friends.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Problem Definition

Software testing is an integral part of the software development process. Test automation makes it more cost-effective and reliable. Automating the execution and validation of test cases is the state-of-practice in the software development process. However, developers still write most of the test cases manually, which is time-consuming and error-prone. Automation of test case generation has received much research attention in recent years. However, in comparison with manual approaches, current test case generation tools generate large test suites which makes them almost impossible to be executed and validated entirely and regularly in the development cycle. Therefore, it is necessary to have an approach that let us automatically order/rank test cases based on their likelihood to detect failing behaviors. This can be beneficial in many testing problems such as fail/pass test oracle problem [8], test generation [31, 82], or test prioritization [42, 70].

One of the most common test adequacy metrics is code coverage, which ultimate goal is to ensure that different source code elements, such as lines, statements, and conditions, are executed at least once during test execution. [89, 54, 1]. It is one of the main targets to optimize when designing an automated test generation [31] or prioritization [42] technique.

Current code-based coverage metrics, however, are not effective in evaluating test cases and suites in terms of identifying bugs. It is not necessarily true that fully covered source code is bug-free; metrics that measure code coverage often miss specification-related bugs [41, 50]. Other alternatives such as fuzzing [90, 35] and input data or test case diversification [16, 25] are also limited in covering software behavior thoroughly since they look at the software under test more like a black-box and do not leverage the underlying source code, execution traces, or specification artifacts. Specification-level criteria, which are defined on the specified requirements (e.g., modeled as a state machine), can help fill the gap. For example, using model-based testing [44, 45] one can represent desired behavior of the system under test and generate and prioritize tests accordingly. However, these techniques are often costly, since they incur the overhead of building models, continuously maintaining them, and the skill set to build, update, and use the models [2].

To tackle the overhead of model-driven software engineering, in general, there has been a lot of progress in reverse engineering different types of models [57, 53]. Creating a behavioral model of a system under test can help identify what parts of the system behavior have been covered by the existing tests and what parts need more testing. However, most of these techniques are not designed with a testing use case in mind. For example, a large body of specification mining work only abstracts a finite state machine (FSM) from the sequence of method call invocations, which is limited since, in testing, input values are often critical and their diversity and coverage can improve testing [28, 26, 46].

The goal of this study is to propose a novel methodology to represent the run-time behavior of test cases that is optimized for testing, and that requires little to no additional manual modeling before use. In other words, a test case representation that can make distinctions between fail and pass behavior (i.e., the ultimate goal of automated testing tasks such as test generation, selection, and prioritization). This requires a rich representation of test cases. For instance, in this study, we emphasize that input and output variables are important in testing and the behavior of the test case should not be represented only using

a sequence of method calls [28, 67, 68]. However, including test inputs and outputs increases the dimensionality of the representation space. Therefore a simple encoding such as the common one-hot-encoding approach is not applicable, and a more advanced model is needed to encode information-rich test execution traces.

A lot of progress has been made in the application of sequence learning in software engineering for both vectorized representation of the system and code generation [6, 30]. Due to the availability of static data, most of these works rely on source code for representing the system. Nevertheless, in software testing, we are more concerned with the system's dynamic behavior than its static characteristics. Recent works have shown that embedding dynamic execution traces of programs is helpful for tasks such as program repair [84, 85] and test oracle generation [81, 80]. S. Zangeneh [88] used classic NLP embedding models, Word2Vec and Doc2Vec, in order to learn the dynamic behavior of test cases by predicting the name of class under test. However, the model was limited to detecting anomalies using diversification methods. Therefore, we decide to improve this model using more recent sequence embedding models and a better learning objective regarding the failing/passing behavior prediction. In the Motivation chapter, we describe how our model improves the original Test2Vec. Throughout this study, we will use the term Test2Vec to refer to Transformer Test2Vec, and we call Zangeneh's proposed model "Original Test2Vec".

1.2 Models and Dataset

In this study, we propose a neural sequence learning model called "Transformer Test2Vec", which can create general representations of test case behavior using historical snapshots of a project. This can potentially be used for multiple downstream testing tasks, similar to how such embedding methods in natural language processing (NLP) are successfully used in several NLP tasks [87, 56]. Our model uses a well-known transformer-based model, called CodeBERT, to learn vectorized representations of sequences of method calls, outputs, and

input parameters, separately. Finally, the model also consists of a BiLSTM layer [34] to act as a classifier and learn the testing-specific characteristics of the sequences, e.g. as pass or fail, based on their similarity to the historical failures. BiLSTM layers are chosen over a simpler classifier layer such as MLP, since they can better learn from the before and after context in an embedding sequence.

To evaluate our approach, we conduct a large experiment on 150 faulty releases from 6 open-source Java projects from the Defects4J [52] benchmark dataset, with a total of $90,556$ execution traces. We use test case prioritization (TP) as a common downstream task to show the practicality and real-world usage of Test2Vec. Our Test2Vec-based TP leverages two heuristics for ranking test cases: (a) the similarity to previously failing test cases, which is motivated from regression testing literature [66] and implemented as a classifier, and (b) the dissimilarity to other test cases in the test suite, which is motivated by test case diversification literature [63, 26] and implemented as an anomaly detection method.

As baselines in our experimental evaluation, we use five different TP techniques from four different categories: (a) two traditional coverage-based TPs as simple approaches representing state-of-practice, (b) a basic trace encoding technique (One-Hot encoding),(c) an alternative state-of-the-art embedding approach (CodeBERT) as the best embedding already applied on other domains in software engineering, and (c) an LSTM-based approach from the literature [81], as the state-of-art method for classifying test case execution as pass or fail.

1.3 Research Questions

We explore three main research questions (RQs) in the experiments. In RQ1, we focus on showing the overall effectiveness of our embedding architecture compared to alternatives, both as simple coverage-based baselines and most advanced embedding architectures. In RQ2, we explore the value of a richer test case representation that includes method calls

inputs and outputs. In RQ3, we look at our downstream task more carefully and explore two potential solutions for the prioritization task that can be implemented by the Test2Vec embedded vectors: classification and anomaly detection. Our research questions and their sub-RQs are as follows:

- RQ1) How effective is Test2Vec embedding for test prioritization when the tests are ranked using a classifier trained on historical data?
  - RQ1-1) How effective is Test2Vec classifier-TP compared to code coverage-based prioritization (state of the practice)?
  - RQ1-2) How effective is Test2Vec classifier-TP compared to a basic and a state-of-art code and execution trace embedding technique from literature?
- RQ2) Does the Test2Vec embedding benefit from input/output parameter values in the execution traces, to better prioritize test cases?
- RQ3) Which of the two prioritization heuristics ("similarity to past failing tests" OR "test diversification") is better to be used with Test2Vec, for a test case prioritization task?
  - RQ3-1) Can changing the diversification technique improve the test prioritization results?
  - RQ3-2) How effective is test diversification using Test2Vec embedding (Test2Vec diversification-TP), for test prioritization?
  - RQ3-3) Can combining diversification and history-based classification (Test2Vec combined-TP) outperform them individually?

To evaluate RQ1, we use both the median and mean normalized rank of the first failing test (FFR) and the well-known average percentage of fault detection (APFD) measures. The FFR metric is measured based on real faults collected from the Defect4J dataset. This

metric measures the rank of the first failing execution trace after ranking execution traces by each method. APFD, however, is calculated based on mutation analysis. The APFD metric essentially measures the weighted average of the percentage of faults detected during the test suite execution. Overall, RQ1 results show that our default TP technique (Test2Vec classifier-TP) leads to significant improvements, in terms of FFR and APFD. The average (and median) FFR relative improvements over baselines in RQ1 are: compared to line coverage 70.93% (72.59%), to branch coverage 69.79% (73.56%), to one-hot encoding model 53.28% (54.16%), to CodeBERT 40.47% (40.62%), and compared to the LSTM-based model 57.92% (60.41%).

The APFD average (median) relative improvements over baselines, RQ1 results, are: compared to line coverage 20.33% (20.72%), compared to branch coverage 22.38% (22.14%), compared to one-hot model 13.79% (12.25%), compared to CodeBERT 1.99% (2.28%), and compared to the LSTM-based model 14.07% (13.83%).

For RQ2, we repeated the RQ1 experiment with abbreviated versions of the Test2Vec classifier-TP. Results show the average (and median) FFR relative improvements over simpler versions of Test2Ve, RQ2, are: compared to Test2Vec(no-output) 21.16% (18.75%) and compared to the Test2Vec(no-I/O) 46.57% (52.50%).

RQ3 is only assessed based on real-faults (using FFR), since diversity-based approaches are not effective when bugs are seeded artificially (i.e., they don't look like anomalies anymore – we will explain this more in the design section). Our results show that in 24% of the times a diversification approach leads to better results than a classifier-based approach. Finally, we show that the ensemble method can improve the results of Test2Vec classifier-TP (from RQ1) by 3.03% and 1.84%, for median and mean of FFR and in 84% of the versions studied Test2Vec combined-TP was better or the same as Test2Vec classifier-TP.

1.4 Contribution

The contributions of this thesis can be summarized as:

- Proposing a novel embedding approach (Transformer Test2Vec) specialized for test case behavior representation that considers the sequence of method calls alongside their inputs and outputs.
- Proposing a test case prioritization method, based on the behavioral similarity of current test cases to historical failures' behavior, leveraging Test2Vec embedding.
- Proposing many diversity-based test case prioritization methods that works as an anomaly detection on the behavior space of a test suite(set of vectors representing test cases using Test2Vec) and comparing them together.
- Proposing a novel test case prioritization method that combines both of the above heuristics.
- Conducting a large-scale experiment, comparing the proposed prioritization methods with both traditional coverage-based approaches, as well assessing the Test2Vec embedding architecture compared to the state-of-the-art code and execution trace embedding models, int he context of test cases prioritization down-stream task.

1.5 Thesis Organization

As for the rest of this thesis, it is organized as follows:

- **Chapter 2:** This chapter discusses motivation for introducing the Transformer Test2Vec model and conducting this study.
- **Chapter 3:** In this Chapter, We review background materials and related works regarding software testing, adequacy metrics, test case prioritization, embedding techniques, and embedding for software testing.

- **Chapter 4:** In Chapter 4, we discuss our approach for trace collection, model architecture, and the process of learning execution embeddings in more detail.
- **Chapter 5:** Chapter 5 covers our data extraction procedure, configurations, and our findings.
- **Chapter 6:** Finally, in chapter 6, we conclude the study.

All the source code and datasets of this study are available in a public repository[1], for replication.

[1] https://github.com/EJabbar/Thesis-2022

Chapter 2

Motivation

Several test case generation tools [31, 82] have been introduced in the software testing community in recent years. However, most of these tools generate tremendous test suites that make the manual validation of the test case executions hard (if not impossible). Many of these tools also need to prune the test cases with a smaller chance of revealing potential bugs in the test case generation process (e.g. tools use the genetic algorithm to generate test cases). These techniques can only be applied, if they have adequacy metrics that can quantify effectiveness of test cases. Therefore, it is necessary that we have an approach to measure the probability of test case fails (which can then be used for test case prioritization or labeling the test cases).

In order to perform most testing tasks, such as prioritizing tests, selecting them, or identifying them as passing or failing, we must first use a method for representing the test cases. The hypotheses of this study is that an execution trace-based embedding can better represent test cases compared to traditional code-driven representations (e.g., code coverage). Essentially, we argue that a good summary of the **dynamic** behavior of the program during testing, as captured by the traces, has rich information and can thus better represent the test cases in later tasks. To quantify "better" on a concrete task, in the evaluation section we use TP as a measurable real-world problem. In this section, however, we motivate our

approach with a simple and intuitive example. This example is based on an artificial code snippet and its test cases.

Listing 1 shows an example function called *formula* that computes $\frac{b^e}{c}$. One of the actual requirements for its sub-function named *power* is that $e >= 0$, otherwise the function should return -1. But this requirement has been missed in the analysis, the code does not implement it, and, thus, the code is faulty. The *check* function is in charge of verifying that all entry requirements are fulfilled. But it contains a bug and does not check whether e is greater or equal to 0.

Assume there are two test cases written for the *formula* function. These two tests have the same branch coverage and statement coverage, while only *test2* fails and triggers the bug. The execution traces (excluding the assertions) for these two tests are as follows:

$$Trace1 : formula(2, 0, 3), power(2, 0), check(2, 0)$$

$$Trace2 : formula(2, -1, 3), power(2, -1), check(2, -1)$$

These tests have the same sequence of method calls with different inputs and only one of them fails.

What we observe from this example is first that the code coverage metrics (at least the branch and statement coverage used here) are not enough to make distinction between a failing and a passing test case behavior. Also, we see that the typical behavior representation in the testing domain, which is a sequence of method calls (that is what most related work in testing and specification mining use), will also, like coverage-based representations, be ineffective (zero distinction between pass and fail). Therefore, a better representation is called for, for instance by considering both the sequence of method calls and the input values. Potentially, additional information can also be utilized such as output values.

Motivated by related work such as code embedding approaches, which are, however, limited to source code and not execution traces [6], and neural fuzz testing, which is limited to

Listing 2.1: A code snippet and its failing/passing test cases.

```
public double formula(int b, int e, int c){
        int p = power(b,e);
        if (c > 0)
                return p/c;
        else
                return -1;
}

public int power(int b, int e){
        int r = 1;
        if check(b,e)
                while(e>0){
                        r = r * b;
                        e = e - 1;
                }
        return r;
}

public int check(int b, int e){
        if(b==0)
                throw new Exception("Undefined");
        return 1;
}

//Tests
public void test1(){
        assertEquals(formula(2,0,3), 1/3);
}

public void test2(){
        assertEquals(formula(2,-1,3), -1);
}
```

input values for a test case [33], S. Zangeneh [88] introduce a representation, called Test2Vec, that focuses on execution traces (sequence of method calls and their inputs) to better distinguish failing and passing tests. The proposed method, however, had some limitations, which led us to create a newer version of Test2Vec called Transformer Test2Vec. In the rest of this chapter, we discuss the limitations of the original Test2Vec and our solutions.

The first limitation of the original Test2Vec model is that this model uses Word2Vec and Doc2Vec models as the building blocks of its architecture. However, studies [20] showed that Test2Vec tends to underperform even basic NLP methods such as TF-IDF in many downstream tasks. More recent contextual embeddings, such as BERT, on the other hand, have shown promising results on many downstream tasks. Therefore, in this study, we decided to replace the old word embedding models used in the original Test2Vec model with CodeBERT which is a transformer-based contextual embedding technique.

The second limitation of the original Test2Vec model is the model's training objective. This model was trained to predict the name of the class under test for each test case. The chosen training objective, however, is **not** chosen in a way that helps the model learn about the filing/passing behavior of SUT. This motivated us to train the new model on a training objective that is more related to software testing. In this study, therefore, we use failing/passing label prediction as the training objective. In this way, our model will learn representations for test cases that have reacher information regarding the behavior of the test case.

Using class name prediction as the training objective of the original Test2Vec also limited the model to predict failures only by finding anomalies by measuring diversification. Using failing/pass labels in Transformer Test2Vec allows us to predict failures not only by detecting anomalies but also by finding the similarity between each test case and historical failures. In this study, we use similarity, diversity, and a combined approach to detect failures.

Since our approach requires a training set that is collected from historical versions of the SUT to be able to represent a new test, it can only be applied if there are some existing

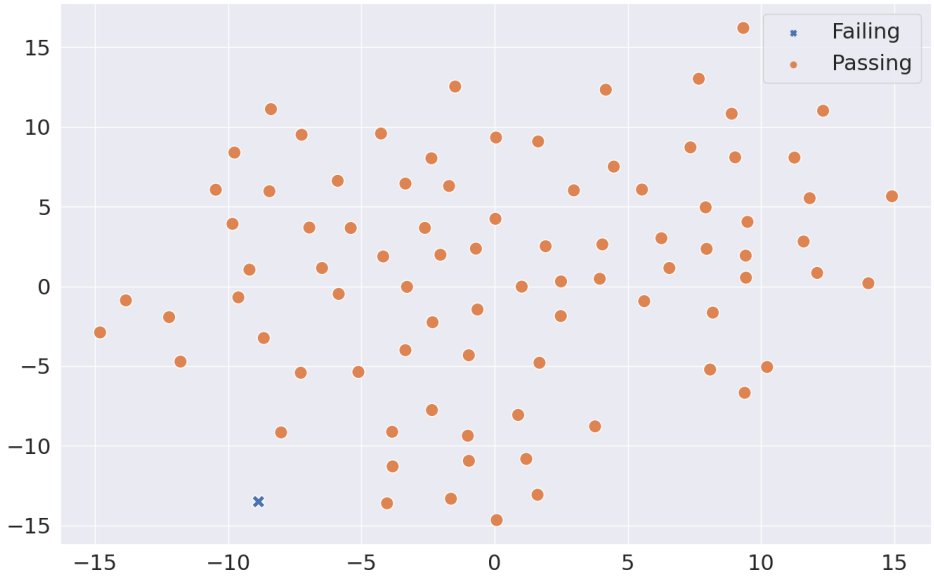


Figure 2.1: Test suite StringUtilsTest of Project Lang61 embedded by Test2Vec and visualized by tSNE. The axes are the latent features learned by tSNE.

tests for a project to start with. Thus, for the example in this section, we cannot run our model. However, Figure 2.1 visualizes a similar example from our dataset (Lang Project V61, in Defects4J), where the failed test's vector (in blue) is significantly different compared to other passed tests that are covering the same buggy code snippet [1]

The above example motivates how our embedding can distinguish the failing test case from the passing ones, which in turn suggests that tasks such as test generation and prioritization can leverage this information. In section 5, we will explore this idea in more detail, in the context of test prioritization.

[1]The visualization is created using our proposed Test2Vec model to vectorize test case traces into an N-dimensional space followed with a tSNE [58] dimension reduction approach to reduce the dimensions into two (for the sake of visualization).

Chapter 3

Background and Related Works

In this section, we provide background needed to better understand our approach and the baselines we will compare it to.

3.1 Software Testing and Adequacy Metrics

Testing is one of the most practical methods of ensuring the quality of software systems. Testing a software ensures that it meets expected requirements. In this process, one or more properties of interest are evaluated using software system components that are executed manually or automatically. As part of the development and deployment cycle, software is tested at different granularities and stages. Automating as much of the software testing process as possible will make it more time/cost-efficient and reliable. Whether we test the software manually or automated, test cases need to be evaluated for quality so higher quality test suites can detect more errors during software testing.

3.1.1 Test Adequacy Metrics

In order to see if a test suite is good enough (adequate) for testing a program, we need to define a metric. An adequacy metric determines if a given test suite, which is designed to test a program, meets its requirements with respect to a given criterion. Test adequacy metrics help testers determine the quality of the test suite and provide criteria for terminating testing. In common practice, adequacy criteria are determined based on the coverage of factors correlated with finding faults, such as the execution of code elements or detecting produced bugs. As an alternative specification test adequacy metrics are defined based on the coverage of the elements in the manually defined models of the system. Nevertheless, neither code-based nor specification-based adequacy metrics can tell us in what order we have covered the software's behavior [40].

3.1.2 Code Coverage-based Adequacy Metrics

Code coverage is one of the most common test adequacy metrics and can ensure that different source code elements, such as statements and conditions, are executed at least once during test execution. It is one of the main targets to optimize when designing an automated test generation [31] or prioritization technique [42]. Most test case prioritization and test case/input generation tools currently use code coverage as their test quality evaluator. Code coverage criteria can be calculated into different granularities, such as:

- **Method/function coverage:** What percentage of the methods/functions defined have been called?
- **Statement coverage:** What percentage of statements have been executed in the program?
- **Branches coverage:** What percentage of the branches of the control structures have been executed?

3.1.3 Specification Coverage-based Adequacy Metrics

Specification-based technique is an automated test generation method that derives test cases directly from the specification (usually represented as formal or semi-formal specification models) which defines what the system should do in each situation. In specification-based testing, testers use abstract models of the system under test (usually generated manually) in order to test the behavior of the system. This method helps them to detect specification-related bugs which cannot be found using code-based testing techniques. However, these techniques are often costly, since they incur the overhead of building models, continuously maintaining them, and the skill set to build, update, and use the models. To tackle the overhead of model-driven software engineering, in general, there has been a lot of progress in reverse engineering different types of models. Creating a behavioral model of a system under test can help identify what parts of the system behavior have been covered by the existing tests and what parts need more testing. However, most of these techniques are not designed with a testing use case in mind. For example, a large body of specification mining work only abstracts a finite state machine (FSM) from the sequence of method call invocations, which is limited since, in testing, input values are often critical and their diversity and coverage can improve testing[28, 26, 46].

3.1.4 Fuzzing

Testing software bugs by randomly feeding invalid, unexpected, or random data into target programs in order to trigger flaws is known as fuzzing [11]. Fuzzing techniques consider the SUT as a black box and create test cases by mutating a set of seed inputs. However, classic fuzzing techniques are limited in covering software behavior thoroughly, since they look at the software under test more like a black-box and do not leverage the underlying source code, execution traces, or specification artifacts. There are more advanced (guided) fuzzing techniques that act as gray or white-box [32]. These fuzzing techniques have been mainly

used for security testing. A broader version of smart-fuzzing is search-based test generation, which defines a heuristic to optimize during the test generation process. A thorough review of all automated test generation techniques goes beyond the scope of this work. In this thesis, we focus on the adequacy metrics and the application of our representation method in test prioritization.

3.2 Test Case Prioritization

Test Case Prioritization (TP) is the problem of ordering the test cases within the test suite of a system under test (SUT), with the goal of maximizing some criteria [79], e.g. executing failing test cases early. TP is mainly used in practice during regression testing and in the continuous integration setups, where new changes to the code should not break the existing features [42]. The traditional TP techniques focus on maximizing code coverage of some sort using an optimization technique such as a greedy algorithm [22, 49] (which is one of our baselines in this thesis).

Another common TP category is fault-based or history-based TPs, where the test cases that have failed in the past (or are similar to the failed tests from history) are ranked higher [66, 24].

Finally, the last common category is diversifying test cases [28], which can be applied on different representations including the specification models [44], outputs [27], and execution traces [28, 63]. The main idea behind this category of TPs is that similar tests cover similar features and behavior. Therefore, diversifying tests should result in a more even coverage of the feature/behavior space. Diversity-based TP can also be seen as an anomaly detection, where the failing test's behavior is considered as an atypical pattern.

3.3 Embeddings in Machine Learning

Embedding is a mapping from an object to a vector of continuous values/numbers. In the context of learning sequences of words, embedding is a mapping from each word to a vector of real numbers that can be used in mathematical models for tasks such as visualization and prediction. With recent progress in word embedding, embedding models can learn about the semantic relation between words in a text and consequently can better represent each word, based on the context [4]. Intuitively, the multiple dimensions of the vector[1] allows the model to express different forms of semantic similarity in different "directions" of the (numerical) vector space.

3.3.1 One-Hot encoding

In one-hot encoding, categorical variables are converted into a form that can be used by machine learning algorithms to make predictions. Using one-hot vectors in word embedding, we distinguish each word in a vocabulary from all other words in the vocabulary by using N (the number of words in the vocabulary) dimensions. By using one-hot encoding, machine learning does not assume that higher numbers are more important. It can, however, result in high dimension vectors that may lower the accuracy of the models generated by this technique. One-hot encoding also has the disadvantage of not capturing semantic relation between words since the distance between all words in the vocabulary is the same.

3.3.2 Word Embedding

In order to solve the issues with sparse matrices generated by one-hot encoding, word embedding techniques have been introduced. Word embeddings are techniques where each word is transformed into a vector (numerical representation) that tries to capture various characteristics of the word in a way that is understandable for machine learning algorithms.

[1] It is not uncommon to use several hundred dimensions and there are even examples using many thousands.

Early word embedding techniques such as Word2Vec by Mikolov *et al.* [60] rely on neural networks to preserve semantic relations in large datasets while computing continuous global vectors. Based on the unique words in a dataset, these approaches build a global vocabulary. Using the other words it tends to appear close to, the representations for each word are then learned. Therefore, semantically similar words tend to be close to each other in the vector space [61]. This model learns embedding for each word by predicting words within a certain range before and after it, given the word itself. One of the interesting characteristics of the embeddings generated with techniques such as Word2Vec is that we conducting algebraic operations on word embeddings we can find a close approximation of word similarities. For example, when we calculate the vec('King') - vec('Man') + vec('Woman') it tends to be close to the vector that the model generates for the word 'Queen'. This makes these embeddings suitable for use in machine learning algorithms. However, learning a global representation for each word means that these models ignore the contextual meaning (a word's meaning derived from its surroundings) of the words. Besides, we just have one representation for a word, but words have different aspects. For example, in the sentence "She left her book in the left locker", only one representation is learned for "left" in the sentence, while it has two different meanings.

3.3.3 Contextual Embedding

As mentioned word embeddings such as Word2Vec cannot capture the context of each word. Alternatively, contextual embedding methods use the whole or parts of a sentence to learn sequence-level semantics. Based on the context of any word, these techniques can learn different representations of it. In other words, contextual word embeddings capture its semantics in context, so even though it is the same word, it can be represented differently under different conditions. Pre-ELMO and ELMO [71] are the first contextual embeddings introduced. These models leverage Bi-LSTM and CNN layers to generate embedding for each token in the context. However, since they use LSTM in their architecture, it is not

possible to apply parallel training for these models. Therefore they are usually trained on small datasets. With the emergence of Transformers, models such as BERT [21], GPT [12], and T5 [74] has been introduced. Using transformers allows the researchers to train these models on huge datasets and generate general embedding models that perform well in many downstream tasks.

3.3.4 BERT Model

In recent years, large pre-trained models have shown promising results on various text analysis problems. BERT is a pre-trained model that achieved state-of-the-art performance on many NLP tasks [21]. It is a transformer-based neural architecture pre-trained on large texts with self-supervised objectives, Masked Language Model (MLM, predict which word was masked out) and Next Sentence Prediction (NSP, predict the sentence that follows). The self-supervision is very important since it allows the use of much larger datasets without manual labeling. BERT considers both the preceding and the following contexts to learn better contextual representations. The success of the BERT model also has led to BERT variants that are pre-trained on specialized corpora [9, 48]. CodeBERT by Feng *et al.* [30] is a BERT variant for both Natural Language (NL) and Programming Language (PL). It learns general-purpose contextual embeddings for both NL and PL. They evaluate CodeBERT on two downstream NL-PL tasks, natural language code search and code documentation generation. In this thesis, we use CodeBERT as the state-of-the-art sequence learning model and apply it to our execution traces, as a baseline. Note that CodeBert is trained on static source code and not execution traces, but there is no other pre-trained large language model for execution traces, either. So using transfer learning we fine-tune CodeBert to be applicable to our trace sequences. The intuition is that the CodeBert model will learn a representation based on the code-like nature of the traces (including the method names and parameters etc.). Then the fine-tuning step provides a more accurate representation.

3.3.5 Similarity/Distance metrics and Anomaly Detection

One of the interesting characteristics of embeddings generated with models such as Word2Vec, BERT, and GPT is that we can find a close approximation of word similarities by applying simple mathematical operations to the embeddings. Therefore, we can use distance measures such as euclidean distance and cosine similarity to measure the similarity between embedded objects.

Euclidean Distance

The most common distance measurement is the Euclidean distance. Basically, it measures the distance between two points by the length of the line segment connecting them. It is also sometimes called the Pythagorean distance since it can be calculated using coordinate points and Pythagoras' theorem. Equation 3.1 shows the Euclidean Distance formula.

$$EuclideanDistance = D(X, Y) = \sqrt{\sum_{i=1}^{n}(x_i - y_i)^2} \tag{3.1}$$

In spite of its popularity, Euclidean distance doesn't scale invariantly, and it becomes less useful as vector dimensions grow.

Cosine Similarity

Cosine similarity, as the name suggests, is the cosine of the angle between the two non-zero vectors A and B. The cosine similarity between two vectors with identical orientations is 1, whereas the similarity between two vectors diametrically opposed is -1. Cosine similarity is calculated using Equation 3.2.

$$CosineSimilarity = Cosine(\theta) = \frac{X.Y}{\|X\|\|Y\|} \tag{3.2}$$

With high-dimensional data and when magnitude does not matter, cosine similarity is a popular choice.

3.4 Transfer Learning

The concept of transfer learning refers to the reuse of a previously created model on a new problem [74]. Due to the fact that it is able to fine-tune and then use deep neural networks on problems where there is relatively little data, it is currently popular in deep learning. Essentially, the model is pre-trained on a large dataset and then fine-tuned by additional training on a much smaller dataset, for a specific task. Using these pre-trained models as a start point tends to improve results significantly compared to training models from scratch on the limited dataset. The pre-trained models are usually trained on general tasks, such as language modeling in NLP. As a result of this pre-training, the model will learn general-purpose knowledge that can then be "transferred" and used in downstream tasks.

3.5 Sequence Learning and Embedding in Software Engineering

In software engineering, sequence learning and embedding has been used for tasks such as code completion[76], program repair [86], API learning [37] and code search [36]. The focus of learning in these works is on the source code [3] and their crucial drawback, in our context, is missing the execution information, which is one of the most important concerns in testing.

There are several applications of sequence learning in software engineering in recent years [65, 38, 3, 17]. For example, Harer *et al.* [39] uses Word2Vec on source code to predict possible bugs or security vulnerabilities of the program. While E. Mashhadi *et al.* [59] generates embedding using a CodeBERT model for program repair. Azcona *et al.* [7] use a simple vector representation model to characterize code, and Chen and Monperrus [18] use Word2Vec and Doc2Vec for program repair. In the category of customized embeddings for source code (neural program embedding), Code2Vec [6], Code2Seq [5], Flow2Vec [78],

and CodeBERT [30] are the state-of-the-art. Though these models' underlying architectures defer, all are working on the program source code and miss the execution information of traces. Since CodeBERT is a pre-trained model (trained on a very large code base) and has outperformed the others, we selected it as our baseline embedding technique.

In the context of representing dynamic executions, Henkel *et al.* [47] uses semantic and syntactic abstracted traces from symbolic execution for C projects to train a Word2Vec embedding model. Their method, however, does not include input values in the training data due to the abstraction. Wang *et al.* [84] propose a dynamic neural program embedding for program repair. They show that using traces including the variables has better results than using the variables or the states traces. In a more recent work [85], for the task of learning representation vectors for the method's body, the authors show that a mix of symbolic and concrete execution traces outperforms other existing approaches that ignore the dynamic behavior of the program. These works are among our motivating works that though applied to different tasks but showed that dynamic traces including input data may be beneficial in testing.

3.6 Representation Learning for Software Testing

Using coverage information: Early works in the test case prioritization mostly used coverage data of test cases to represent each test case. Elbaum *et al.* [23] used multiple coverage-based representations (vary in granularity and considering additional or total coverage), in the TP task. They conclude that techniques with finer granules typically outperform others. D. Nardo *et al.* [22] also showed that coverage-based techniques that use additional coverage with more detailed coverage criteria are more effective at detecting faults.

Using text similarity: In this category, each test case is considered as a string by looking at its source code, and the similarity or distance of test cases is measured by applying simple string distance metrics (without applying any encoding or embedding). For instance,

Miranda **et al.** [62] used string distance metrics, such as Hamming distance and Levenshtein diversity, to measure similarity/distance between test cases.

Using static data of test cases: This category contains works that use static data related to test cases such as source code, specification models, comments, and ASTs in order to represent test cases. For example, Hemmati *et al.* [44] encode tests based on specification coverage of each test case on the manually drafted state machine representation of the system. Mondal *et al.* [63] used one-hot encoding to represent each test case based on a set of method calls that are extracted from each test case script, without executing the code. Thomas *et al.*[79] uses a topic modeling approach on the source code and comments.

Using input data: These works usually try to represent test cases based on one or a sequence of inputs. In other words, they are black-box fuzzing approaches that only consider test inputs. For example, Godefroid *et al.* [33] have leveraged neural statistical models to automatically infer input grammars for Fuzz testing.

Using dynamic traces of test cases: Another approach for representing test cases is using dynamic data, such as execution trace. Using dynamic data for software testing is not explored enough by researchers. Noor and Hemmati [69] used dynamic execution trace of the program to represent a test case by applying a one-hot encoding that only consider method calls. MK Ramanathan *et al.* [75] represents each test case with the sequence of memory operations and their values at the run time execution of the test case. In order to find the similarity of tests within a test suite, these sequences are then used to create a dissimilarity graph. Tsimpourlas *et al.* [81] used recurrent NN and execution trace of the programs for classifying test cases as pass or fail. We used this approach as one of our baselines in this thesis.

Chapter 4

Methodology: Transformer Test2Vec

The key idea of this thesis is a new embedding model for dynamic execution of test cases. The idea is that after applying our embedding on each test case execution (execution trace), that test's run-time behavior is mapped to and thus is represented as a vector in an N-dimensional latent space. These embedded vectors can then be used to analyze a test suite as well as individual, and groups of, test cases. For instance, using them one can predict whether a test case will pass or fail, using a simple softmax classifier and the actual, historical test outcomes (as target labels). Alternatively, a test case with an anomalous, i.e. different, behavior within a test suite can be detected by calculating the distance between such vectors in the latent space (vectorized representation) of the test suite. Much like how embeddings have been used for natural language processing in the deep learning literature, test embeddings can be the basis for many different, downstream tasks in automated testing.

Thus, we propose "Transformer Test2Vec Embedding" (or in short Test2Vec), a neural embedding model leveraging pre-trained models, to vectorize test execution traces. The inputs to our embedding model are traces (sequences of method calls, their input parameters, and their outputs). The trace embedding model is fine-tuned based on the corresponding test case's pass/fail label per execution trace. There are other options for fine-tuning the model, such as class name or method name prediction. However, since we are more interested in the

failing behavior prediction, for our particular downstream task (TP), we argue that using the pass/fail label as the final objective for fine-tuning makes more sense than the other options. It should be noted that since we need to run the test cases in order to collect the execution traces, we already have the pass/fail labels for each test case in our historical versions of SUT. Therefore, there is no overhead for collecting the pass/fail labels in comparison with other options such as class or method name.

The specific Test2Vec implementation we propose and evaluate here uses CodeBERT layers that are originally pre-trained on generic tasks, such as masked label prediction, on both programming and natural language data. CodeBERT's job in Test2Vec is to embed methods and I/O parameter names, individually, which is the closest sub-task within trace embedding that can leverage the learned representation from CodeBert's pre-trained model on static source code. However, to be useful for execution trace embedding (vs. the original static code embedding) these layers are then fine-tuned, end-to-end, while we are training the Test2Vec model for test case prioritization that let the model learn the passing/failing behaviors for the SUT. In this study, our fine-tuning labels are pass/fail labels that are collected by Junit execution of the test cases, but in general, our Test2Vec method can be applied to a variety of testing-related tasks with potentially a different fine-tuning.

Our hypothesis (which we will verify by the results of the experiment) is that a project's dataset including the current revision's test cases as well as historical testing results from past revisions will be enough for fine-tuning the model. The assumption is that these trained models will help us with e.g., predicting which test cases are more likely to fail by classifying their traces based on the historical failures of similar traces in the past. It will also help analyze an individual test case's embedding, in relation to other test cases in the current test suite. For instance, the embedding vectors of similar test cases within a class, which are all covering the core functionality and have a lot of overlap, should become closer to each other. Whereas, the vector representation of a test case covering the less tested feature, within the class, becomes more like an outlier in the test suite (more distant from the other vector

representations). In addition, anomalous run-time behavior resulting from executing a failed test case will also be likely represented as an anomalous vector, within the behavior space (set of vectors in a test suite). However, critical in this is that the sequences of methods called might not be enough to capture fine-grained differences in behavior. Modeling also input and output values can help in better distinguishing test case behavior.

The definition of an execution trace in this study is, thus, a sequence of method names (all methods of the class under test that are invoked during execution of the given test case) with their inputs and outputs. The choice of what to include in the trace is somewhat arbitrary and a systematic study is needed in the future to identify the elements that contribute the most and thus should be included, per task. However, the method calls, their inputs, and their outputs seem like the minimum that is suggested in various literature [81, 29, 43].

We call each triplet of $<Outputs, MethodName, Inputs>$ a *Context*, since each method-call is, in fact, a context for other method-calls within the whole sequence. Given that each trace can have a varying number of method invocations (Contexts), a challenge is to map all of these Contexts into one single embedding vector.

Figure 5.1 and Figure 4.3 show an overview of our approach, which consists of three main phases, as follows:

- Executing the test cases and collecting the traces
- Generating a single vector representation for input sequences of each test case
- Learning the fail/pass behavior of the execution and fine-tuning the vector representation models

In the rest of this chapter, we are going to explain each step in detail.

4.1 Phase1: Test case execution and trace collection

The process starts with running all existing test cases per project (in our case, the developer-written unit tests) for the current version of the SUT and its historical versions. That means our dataset for each project is based on the current and historical test cases within the same project. Historical test cases are each run on their corresponding source code versions (since most older versions have obsolete test cases that are not executable or even compilable, in the most recent version of the program). This will generate traces for all developer-written test cases including failed test cases in all of the target versions. However, since normally there are many more passing tests than failing ones, this will lead to a highly unbalanced dataset with hundreds of passing traces and a few failing traces.

To mitigate the unbalanced dataset problem, we use mutation tools to automatically seed artificial faults and generate more failing test cases. This will help us to have more failing traces for the current version of the SUT and its historical versions, and we will have a balanced dataset. The overall benefits of this approach are as follows:

- **Balanced dataset:** Models with a balanced data set would be more accurate, provide a higher detection rate, and result in more balanced accuracy.

- **Covering more failing behaviors:** By using mutation, we can increase the size of our dataset and cover more failing behavior. This will increase the model accuracy since it can learn more failing behaviors for different modules in the SUT and have a better prediction for future failing behaviors (that never happened in the original historical versions of the program).

To log the execution traces of the test cases, we have instrumented the SUT code that allows us to capture the method calls (including nested calls when the SUT is called) and all I/O names, types, and values at run-time.

As mentioned, each execution trace includes the sequence of invoked method calls while running the test case. This means we are not limited to the method names which are in the

Table 4.1: List of mutation operators used by Major framework to mutate PUTs.

Mutation operator	**example**
Arithmetic Operator Replacement	$a * b \rightarrow a/b$
Logical Operator Replacement	$a\&b \rightarrow a\|b$
Conditional Operator Replacement	$a\&\&b \rightarrow a\|\|b$
Relational Operator Replacement	$a <= b \rightarrow a < b$
Shift Operator Replacement	$a >> b \rightarrow a << b$
Operator Replacement Unary	$-a \rightarrow !a$
Expression Value Replacement	$a = b \rightarrow a = 0$
Literal Value Replacement	true $\rightarrow$ false
Statement Deletion	break $\rightarrow < no - op >$

test script and can include all nested calls, invoked at run time. However, to keep each trace focused on the Class Under Test (CUT) if a method of the CUT invokes methods belonging to another class, only the first method invocation from the external class is included (the method that was called from CUT) in the trace and the rest of calls are ignored. Essentially in this way, we isolate units–CUTs– and focus on unit testing rather than integration testing.

The execution trace also includes the method's input parameters (if any), outputs, and their data types. Therefore, our traces may contain a lot of numerical data (alongside textual data). Research [83] has shown that most NLP techniques cannot work well with numerical data, especially large values. They also demonstrated that the state-of-art NLP models, such as BERT, that rely on sub-word tokenizers have difficulty embedding large numeric and float values. For this reason, we tried a simple abstraction technique for primitive types that replace the value of each primitive parameter with one of the predefined special tokens, based on its value. For $int, short, long, double, float$, we convert the numerical values to an abstract format that keeps the range of value but not the exact value. This way although we will lose some information (the exact input value), we keep the information that is understandable for the neural model. Besides, for the testing purpose usually, the exact value of the input may not be that important, and faulty behavior usually appears in a range of values.

For *string* and *array*, we just abstract into empty vs non-empty string or array. For

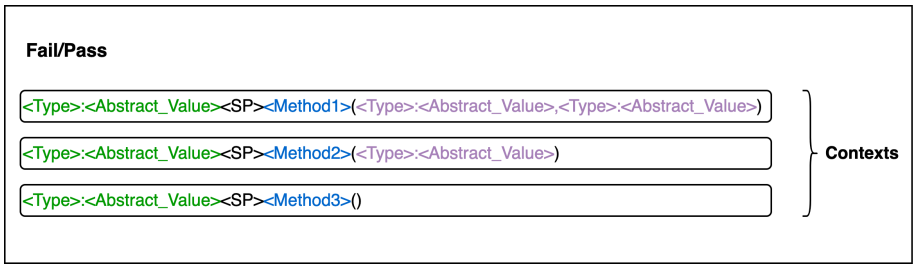


Figure 4.1: Trace template after pre-processing.

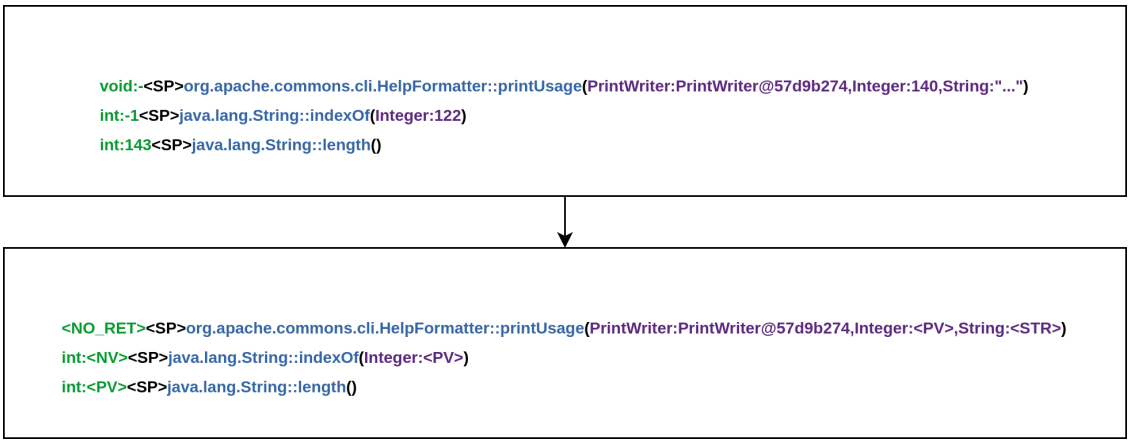


Figure 4.2: A real example of execution traces passed to Test2Vec, before and after prepro-cessing

non-primitive parameters (objects), we include their object type and their reference value. Please note that this approach for abstraction is only one plausible way and there is still room for more research in this area, to determine how to best handle numerical data and complex objects within execution trace embedding, which is in our future work. Figure 4.1 shows an example of a method called after abstraction.

Figure 4.1 shows the template of the collected traces after prepossessing, which includes a sequence of tuples following this format: *Test Result, Output Type, Output Value, Method Name, Parameter Type, Input Value*, in which *Test Result* shows whether the test case has been passed or failed.

Figure 4.2 shows an example of raw execution trace and its representation using the template.

In practice, the execution traces typically vary quite significantly in length. To have a consistent, limited size, and less sparse training data, we have limited the number of contexts

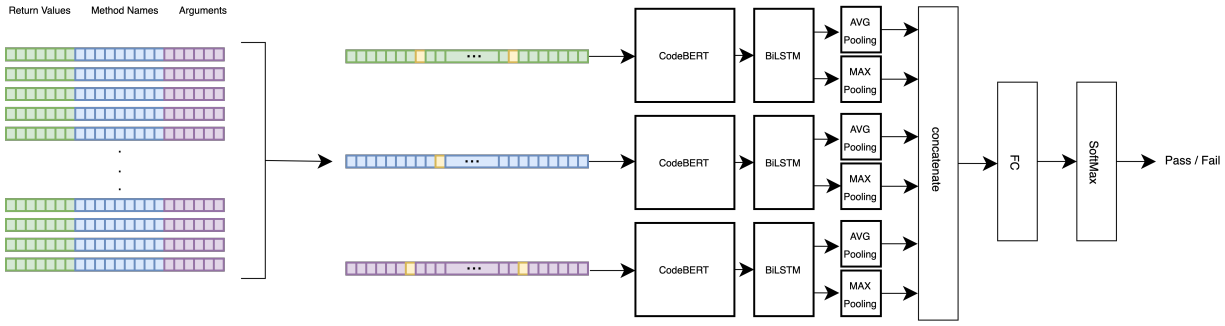


Figure 4.3: Test2Vec Overall Model Architecture.

in a trace (number of calls) to MAX Context. Besides, the CodeBERT model accepts a limited number of tokens (in our study, we used the maximum number of tokens, 512 tokens, which it can process). Therefore we limited the number of contexts for each trace to 128, by excluding the contexts from the middle of each long sequence, since these contexts for long trace sequences, tend to be repeated calls in loops. This value is such that most of the traces will have less than 512 tokens for each CodeBERT model, and more than 75% of traces remains unchanged.

4.2 Phase2: Trace embedding generation

In order to generate a single embedding for each context (sequence of method calls and their corresponding inputs and outputs), we perform the following steps:

- extracting three sequences for each context: (1) sequence of method names, (2) sequence of inputs, and (3) sequence of outputs.
- generating embedding for each token in sequences.
- generating a single embedding for each sequence that represents each sequence.
- concatenating the three generated vectors in order to have a final embedding for each context.

This subsection describes each step in detail.

extracting the sequences

After extracting the context for each test case execution and applying the value abstraction in step one, we extract three sequences. To generate the method name sequence, we simply extract method names from the context (preserving their order) and put them in a single sequence separated by a special token ('NXT').

For input and output sequences, we extract the I/O types and abstracted values for each method call. Since a method call may have more than one input, we separate them by a token ('NXT-ARG'). For methods with no input or output value, we use the ('NO-ARG') token. In the end, we concatenate them using ('NXT') token as the separator to generate the final sequences for inputs and outputs.

token embedding generation

To generate a vector representation for each token in the sequences, we pass each sequence to a distinct CodeBERT model to generate a separate embedding, per token, in each sequence. The CodeBERT layers start with the pre-trained weights of the original paper. However, each layer will be fine-tuned to generate better representations for each sequence. In other words, to fine-tune each CodeBERT, we start with the weights of the pre-trained transformer models in CodeBERT and then fine-tune the models on our training data (traces from 20 versions of a project), while classifying the traces as pass or fail.

sequence embedding generation

We use a small LSTM layer after each Transformer, which is a very common way to extract sentence embeddings from BERT models [64]. At this point, we still have a separate embedding for each token in each sequence. We then use pooling layers [19] to generate the final representation for a sequence and aggregate these token embeddings. We use both Max and Average Pooling layers and concatenate those to generate a final embedding for the sequence.

final embedding generation

In the end, In order to have a single embedding for each context that contains information

regarding the sequence of method calls and their I/O, we simply concatenate the three vectors that we calculate for the sequences using a concatenation layer in our model. This vector is considered the final trace embedding. We can use these embeddings as an input for a classifier or by applying anomaly detection techniques.

4.3 Phase3: Training the classifier and fine-tuning the embedding models

After generating a single representation vector for a test case, called *Test Vector*, we use this representation alongside the label of the trace, the passing or failing label that we collect in Phase 1, to train the end-to-end model (including fine-tuning the CodeBERT layers). In this way, we can have embeddings that capture more information regarding the fail/pass behavior of each trace and a classifier that predicts the probability of being failed or passed for each embedding (learn a conditional distribution: $P(Failing|Embedding)$).

The training of the model is done in two steps. We, first, freeze the embedding generation layers. This is necessary because without freezing these layers, weights of these layers will be updated from the first epochs, and since the classification layers are not accurate at the beginning of the training process, the pre-trained weights start to update randomly, and we will lose the benefits of transfer learning. Therefore, we started the training process by freezing the CodeBERT layers and only training the rest of the model.

After training the LSTM and the classifier layers, we start the CodeBERT fine-tuning process. For fine-tuning, we start with unfreezing the embedding layers and reducing the learning rate of the model. This will help the model update weights for the embedding layer without altering them too much and forgetting the knowledge it learned during the pre-training process. We then resume the training for a maximum of ten more epochs. This will let us update the weights of the whole model end-to-end.

After these three phases, Test2Vec provides us with a representation per test case that

is fine-tuned based on historical failing test cases. In the next section, we evaluate this representation for a test prioritization task.

Chapter 5

Evaluation

In this section, we describe the details of the experimental evaluation including the specific research questions, the design of the experiments, as well as the findings.

5.1 Hypotheses and Research Questions

We investigate three hypotheses in three main research questions as follows:

RQ1) How effective is Test2Vec embedding for test prioritization, when the tests are ranked using a classifier trained on historical data?

The first hypothesis of this study is that Test2Vec allows a fine-grained representation of test cases which can improve performance on downstream tasks, with respect to failing/-passing behavior, compared to code coverage and state-of-art code and trace embedding techniques. To validate this hypothesis, in RQ1, we compare Test2Vec with alternative representation methods for a specific downstream task of test case prioritization (TP). We argue this is both an important, well-studied, and representative task in automated testing. Our default approach (Test2Vec classifier-TP) for test prioritization based on Test2Vec is using the classifier (softmax) layer's probabilities that we trained for the pass/fail label prediction task. To address RQ1, specifically, we answer these two sub-RQs:

RQ1-1) How effective is Test2Vec classifier-TP compared to a code coverage-based pri-

oritization (state of the practice)? The goal of this sub-RQ is to justify the whole idea of test case behavior representation compared to simple code coverage, for a task such as TP. As a basic coverage-based TP, we use the Greedy-based additional coverage TP [49], which ranks the test case that covers the most uncovered lines/branches higher, at each step. This baseline has been used in much of the TP literature as coverage-based baselines. Our proposed approach will be Test2Vec classifier-TP that ranks test cases using a softmax classifier trained on historical data. This classifier calculates the probability of being a failing test for each trace that is used for ranking test cases (simply by sorting the traces descending based on their failure probability).

RQ1-2) How effective is Test2Vec classifier-TP compared to basic and state-of-art code and execution trace embedding techniques from literature? The goal of this sub-RQ is to justify the use of our proposed customized and relatively advanced embedding compared to existing code- and test-based embeddings. To do so, we compare Test2Vec classifier-TP with test case prioritizations calculated with classifiers that use (a) one-hot-encoding as a simpler embedding, (b) CodeBERT, as the state-of-art code embedding model, and (c) an LSTM-based model, as the state-of-art execution trace embedding model.

RQ2) Does the Test2Vec embedding benefit from input/out parameter values in the execution traces, to better prioritize test cases?

Our second hypothesis is that considering the inputs and outputs is important to capture behavior of the model. In RQ1, our Test2Vec embeds all the method calls and their input/output values. Given that most existing techniques ignore input and output values and including them brings some challenges for encoding, e.g., the high-dimensionality of the behavior space, here we do an ablation study to understand the impact of including these values. To do so, we create two new baselines called Test2Vec(no output) and Test2Vec(no I/O), which keep the embedding architecture untouched compared to RQ1, but exclude the input and output parameters from the embedding.

RQ3) Which of the two prioritization heuristics (similarity to past failing

tests or test diversification) is better to be used with Test2Vec for a test case diversification task?

Our third hypothesis is that the representational power of the test case embeddings we propose can be leveraged with diversification as a natural, performant, and potentially complementary heuristic for the TP problem. Since diversity has shown promise in prior test automation and prioritization work [28, 26, 46] a rich representation of execution traces might benefit such prioritization methods. To leverage this heuristic, we used an algorithm that ranks a set of test cases iteratively, by prioritizing the most diverse test case (i.e., the most dissimilar test case compared to others in the latent, embedding space), in each iteration. While this is not the most efficient way this can be done our main goal is in evaluating the power of the embeddings; future work can investigate more efficient approaches. In order to examine the effectiveness of this heuristic, we designed the following sub-RQs:

RQ3-1) Can changing the diversification metric improve the test prioritization results? In this sub-RQ, we investigate whether choosing different diversity measures can impact the performance of Test2Vec diversification-TP. To do so, we repeated the Test2Vec diversification-TP using the same model architecture and TP algorithm, but with different diversity measures (distance to the centroid, average minimum distance, and COV measure) to identify anomalies in the test suites.

RQ3-2) How effective is test diversification using Test2Vec embeddings (Test2Vec diversification-TP), for test prioritization? In this sub-RQ, we compare the results of our default TP (Test2Vec classifier-TP) from above with a diversification-based one (Test2Vec diversification-TP). Given that the two heuristics potentially target different types of fault, we also consider if the approaches can be combined by a classifier that selects which method (classifier or diversification) should be used. We call this method Test2Vec combined-TP.

RQ3-3) Can combining diversification and history-based classification (Test2Vec combined-TP) outperform each of them individually? The hypothesis of this sub-RQ is that the bugs that are similar to historical failures can be better ranked by default TP and those that are

new and pose as anomalies (within the existing tests) can be better ranked by diversification-TP. We also argue that if this is the case it adds further support that the embeddings we propose can capture rich information about test case behavior. Thus, it makes sense to propose a combined TP that dynamically figures out which approach is better suited per test suite. Therefore, we propose the "Test2Vec combined-TP" approach, which is a classifier that is trained on the historical version of the project to predict the better heuristic per test suite, given some features from test vectors and their labels. RQ3-3 compares Test2Vec combined-TP with its constituent alternatives when used individually.

5.2 Evaluation Metrics

Similar to representation learning literature in NLP [77], evaluation of a representation method is more meaningful through a downstream task. In our case, we chose TP as our downstream task given that it is one of the main use cases of automation in real-world software testing and has been well studied in the literature. Another reason is that unlike automated test generation, which depends on multiple factors (e.g., the abstract test representation and design, the input data generation strategy, and approaches to make executable tests out of abstract tests, etc.), automated TP depends heavily on fewer factors, i.e. the test representation and the ranking strategy. Thus, we vary both of the latter in our experiments.

The evaluation metrics we have used are standard metrics from the TP domain, such as APFD [70] and the rank of the first failing test [14] (FFR). The FFR metric represents the normalized position of the first failed test case for a test case prioritization method. Equation 5.1 defines FFR more formally.

$$FFR = 100 * \left(\frac{TF_1}{n}\right) \tag{5.1}$$

where n denotes the number of test cases to be prioritized and TF_1 is the number of tests that must be executed before the first fault is detected by the prioritized tests. Note that we

normalize the ranks based on the number of test cases since otherwise a TP's effectiveness in ranking test suites A and B would be the same when it ranks their failing tests as #5 in both A and B, even if in A there are only 10 test cases (the TP found the failed test only after running half of the test suite) and in B there are 100 tests (the failed test was detected among top 5%).

Our second metric, the average percentage of fault detection (APFD), captures the average percentage of faults detected by a set of prioritized test cases. It can also be seen as the Area Under Curve where the x-axis is the ranked test cases and the y-axis is the cumulative number of faults detected when the tests are executed following the x-axis order. The higher the APFD, which ranges from 0 to 100, the better the fault detection system. Equation 5.2 defines APFD more formally:

$$APFD = 100 * \left(1 - \frac{TF_1 + TF_2 + ... + TF_m}{nm} + \frac{1}{2n}\right) \tag{5.2}$$

where n denotes the number of test cases to prioritize, m is the total number of faults detectable by these test cases, and TF_i is the number of tests that must be executed before fault i is detected. APFD is more informative if there are multiple choices of failing test cases and faults to prioritize so that the area under the curve is a good "summary" of the prioritization effectiveness. Otherwise, if for instance, there is only one bug and one failing test, simply measuring the rank of the failing test is a more direct and meaningful metric.

Because of the characteristic of our dataset (Defect4J), which mostly has only one failing test per bug and only a few bugs per version of the project, APFD is not suitable on Defect4J's real bugs. Therefore, we only use the rank of the first failing test as our evaluation metric when dealing with the real bugs. However, we also injected seeded faults using mutation testing into the projects to be able to compare techniques with respect to ranking seeded bugs. Since we have many failing test cases, in this case, we can use APFD to compare models.

Besides the choice of metrics (FFR vs APFD) another design choice we have is at which

level to apply such metrics (i.e., whether to prioritize test cases of one class or the entire project's–all test cases of a version combined). The two approaches can be debated. Applying TPs to one class is more accurate and provides better rankings since we only compare behaviorally similar test cases. Comparing all test cases from all classes are sometimes very noisy. However, to begin the TP process at the class level, one must know which class to focus on.

To deal with this challenge we evaluate our approach on both levels. In the whole version level, we use APFD to summarize the effectiveness of the TP over all test cases and faults (i.e., both real and seeded faults). To evaluate the TPs at the class level, we use the FFR (normalized rank of real faults). In this way, we get a sense of TPs' effectiveness at both levels. However, our main metric is still the FFR on the class level for the following reason: (a) It is the most direct metric for ranking, in particular when there is only one fault to catch. (b) We use it on the real faults. (c) As we see later in this section, APFD on mutants does not work on our diversity heuristic. (d) In practice, there are other ways to decide which source classes should be tested (e.g., using defect prediction, or simply based on the last changes to the code) and thus a TP technique can know in advance which test suite(s) to target.

So in summary, whenever we report AFPD results they are the summary of TP's effectiveness on ranking all test cases in all test suites, per version, to detect mutants. Whereas, reported FFRs are the results within the failing test suite (only test cases of one class), with respect to catching real bugs.

5.3 Dataset and the Data Extraction Procedure

In this study, we run our experiments on 6 open-source Java projects from the Defects4J project [52]: Commons-Lang (Lang), Commons-Cli (Cli), Commons-Compress (Compress), Jsoup, Mockito, and commons-math (Math). Defects4j provides multiple revisions per

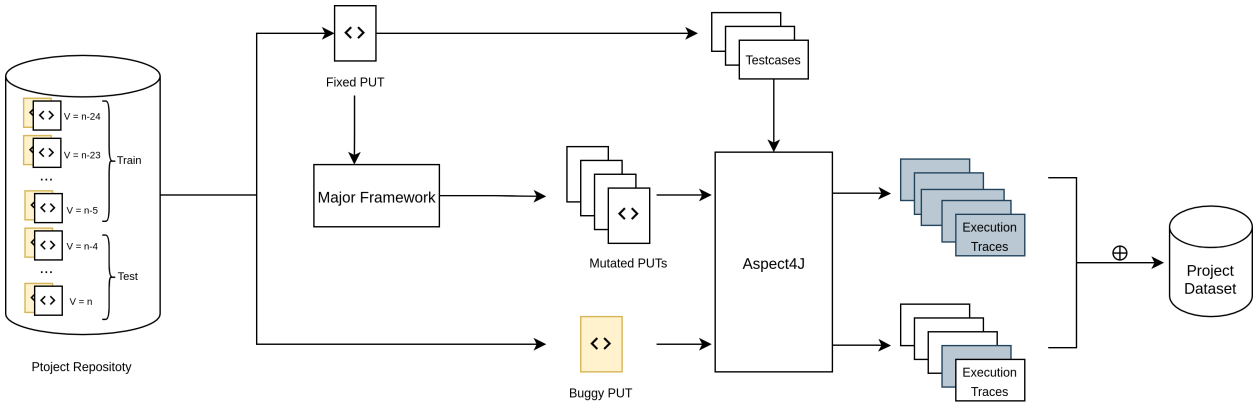


Figure 5.1: Overall process of generating the trace datasets, per project. "failed" traces (shown as colored boxes) are collected from running tests on mutants or real bugs.

project, each one providing a faulty and a fixed source code version. For each project, we used the last 25 revisions (20 older revisions for training and 5 most recent revisions for testing). Table 5.1 shows the number of versions and classes per project.

In Defects4J, each faulty version has exactly one bug (failure) and one corresponding failing test suite (called "triggering test suite"), with at least one failed test case (called "failed test"). With respect to seeded faults, as it is the norm for mutation testing, we make many mutants per version, where each mutant isolates one fault. In most versions of the projects, there is only one failing test case in the failing test suite, but there are also versions that have multiple failing test cases.

Figure 5.1 shows the process of generating traces for each project in Defect4J. First, for each version of the project, we fetch its buggy version and run all its corresponding test cases. In our dataset, this will generate a set of traces that are mostly passing traces and only a few failing test cases (usually one or two failing test cases). To make the dataset balanced, we then fetch the fixed version of the program and give it to our mutant generation tool, the Major [51] framework. The Major framework will generate a great number of mutated versions of the source codes that are used as the Program Under Tests (PUTs). For mutation generation, we used the common mutation operators (the list of the operators we used in this experiment can be found in Table 4.1). We run the test cases on the PUTs and collect traces for only the failing test cases. We repeat this procedure, by randomly selecting a new

PUT and running the test cases until we have the same number of failing and passing traces for each version of the program. To avoid the problem of equivalent mutants, we ignore PUTs with no failing test cases.

As discussed, we augment our dataset with the failing traces on seeded faults, until we have an exactly balanced dataset. Given that the total number of failing tests on seeded faults is more than needed (to keep the balance between pass and fail) we have to select a subset of failing traces. Therefore, we randomly select only one failing trace per seeded fault, so each trace covers a unique fault.

Since the method calls related to assertions or exceptions may contain information about test case results in their input parameters or outputs, we remove all the method calls related to exceptions and assertions from the traces in order to prevent data leakage.

We ran all developer-written test cases of the projects and aggregated all traces from all faulty versions and mutated versions of a project, to create a dataset per project. It should be noted that some of the tests might be repeated across versions of a project. However, since the tested source code of different versions is not always the same, we kept the repeated tests as their traces might still differ.

In the end, we had a total of 90,556 execution traces (45,278 traces from faulty versions of the project and 45,278 traces from mutated versions) over the 6 projects, ranging from around 2,186 to 15,732 traces, per project. Table 5.1 reports the number of traces per project as well as the distribution of traces in the failing test suite, per project and failed class.

We used Train and Validation datasets for training and evaluating the Test2Vec embedding model, and the Test dataset for analysis of the downstream task (TP). Thus, each project has its separate dataset and trained model. We then split each project's trace dataset into Train, Validation, and Test sets. Among the last 25 included version per project, we use the last five versions (21-25; where V25 is the most recent version) as the Test set. For the other 20 versions, we aggregate all the traces, including traces related to mutated source codes. We, then, split it into Training (80%) and Validation (20%) sets.

Table 5.1: Projects Under Study.

Project	#Total Traces	#Trace Per Failed Class Min	Median	Max
Cli	2994	18	23	170
Jsoup	6798	6	23	132
Compress	8352	7	11.5	17
Lang	15732	11	19	73
Mokito	2186	7	8	9
Math	9216	7	11	16
Total	45,278			

5.4 Design

To answer **RQ1-1**, we implemented two TP methods: (a) an Additional Greedy Algorithm [42] that maximizes a given coverage (in our study line and branch coverage) and (b) a Test2Vec classifier-TP Algorithm that ranks traces based on the $P(failing|trace)$ calculated by a softmax classifier that ultimately shows the probability of a test case failure based on the similarity of its trace to historical failures.

Coverage-based TP: Since our focus is on the benefits of our proposed embedding approach, rather than on the optimization step of TP, we use a Greedy optimization algorithm for both TP methods, mentioned above. Future work can replace the Greedy algorithm with alternatives like meta-heuristic methods.

The additional Greedy Algorithm, applied on line (branch) coverage, simply starts with the test case with the highest line (branch) coverage. Then, the next test case that covers the highest "uncovered" lines (branches) will be added. This will continue until all test cases are ordered. Whenever there is more than one test with the same best coverage, one is chosen randomly. JaCoCo was used to collect the line and branch coverage values per test case.

Test2Vec classifier-TP: To calculate Test2Vec classifier-TP, first, we train the "Test2Vec model" for each project using its training dataset (which includes sequences of method calls, their outputs, and their input values). Since we train the Test2Vec model for classifying execution traces to fail or pass, we can use the same probability that is calculated by the

softmax layer for ranking the trace executions, as well. In other words, these probabilities show the confidence of the model about an execution trace belonging to a failing test case and thus being prioritized in the TP process.

The goal of **RQ1-2** is to compare Test2Vec with the current body of research [30, 81] in execution trace and code embedding techniques, for the TP task.

Our first baseline in this sub-RQ is a basic One-Hot encoding. In this approach, we simply extract all the method calls (that are called from the test case and class under test) from the trace and create a one-hot representation of these method calls. Since there are many possibilities for the input parameters and output values, it is not possible to use one-hot encoding to represent them. Therefore, we ignore input parameters and outputs in this approach. The output of this technique is a sparse diverse encoding for each trace. We then pass these encodings to a classifier to be classified as pass or fail. The main reason for this baseline is to understand whether including I/O and consequently being forced to use advanced embedding is necessary or a simple one-hot representation is enough for execution trace encoding for TP.

Our second baseline in this sub-RQ is the most related work (we call it the LSTM-based trace embedding) in the domain of execution trace representation for testing [81]. LSTM-based trace embedding uses encoded arguments and return values and concatenates these representations along with one-hot representations of the caller and callee methods to generate the same size vectors for each method call in the trace. These vectors then will be passed to an LSTM and MLP (Multi-Layer Perception) to be classified as pass or fail.

The last baseline in this sub-RQ is the state-of-the-art neural program embedding model, **CodeBERT**. The idea here is we use one CodeBERT to embed the entire execution trace and do not bother with the customized architecture of Test2Vec (which uses CodeBERT but adds extra structure and layers to it; See Figure 4.3). Since CodeBERT is a generic embedding model, we need to at least fine-tune the model for the TP task. To do so, we take the final fixed-size embeddings of each trace from the CodeBERT using maximum and

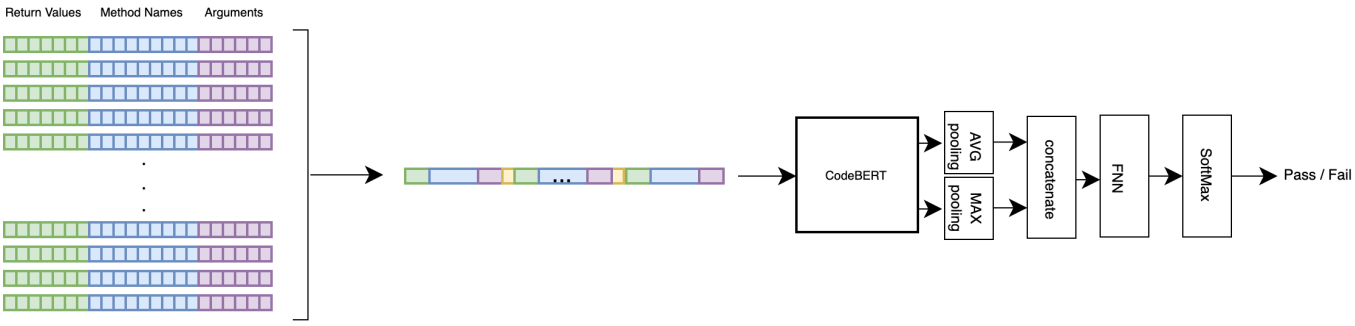


Figure 5.2: CodeBERT-TP Overall Model Architecture.

average pooling strategies and concatenate these vectors together (see Figure 5.2). These embeddings are then passed to an MLP layer and a softmax classifier to predict the final label.

For the actual test prioritization in both LSTM-based and CodeBERT TPs, we apply the very same process as Test2Vec classifier-TP, i.e., ranking tests based on their softmax classifier's probabilities.

In **RQ2**, we keep the Test2Vec model as much as possible untouched and only exclude the embedded vectors of the inputs and outputs from the final vectors (i.e., set the size of (a) outputs and (b) inputs and output embeddings to zero and keep the other configurations of the model, including the final embedding size, the same). These baselines (**Test2Vec(no-output) classifier-TP**) and (**Test2Vec(no-I/O) classifier-TP**) is then compared with the original Test2Vec classifier-TP. Note that the methods' names and their input/output values are one of many potentially relevant information to embed (but perhaps most important ones) and future work should more systematically explore other factors [28].

In **RQ3-1**, we compare different diversity measures for the task of test case prioritization using the diversification method. We compared three following diversification measures:

Distance to the centroid: This technique measures the distance between each vector representation and the centroid of the vector representations of the test cases in the test suite and ranks them based on this distance. Algorithm 1 shows the overall procedure of this technique.

Algorithm 1 Distance to the centroid algorithm [88]

Data: Dataframe of (Test Name, Embedding Vector)
FT_Name: Name of Failed Test Failed test's normalized rank

```
procedure ComputeRank(Data, FT_Name)
    centroid ← mean(Data.vectors , axis = 0)
    distances ← cosine_distances(Data.vectors , centroid)
    Data.add(distances)
    Data.sortValues(by =′ distance′, Descending = True)
    Rank ← Data.index(FT_Name)
    Normalize_Rank ← normalize(Rank, Data.length)
    return Normalize_Rank
end procedure
```

Average Minimum Distance: This technique ranks test cases based on the average distance of the vector representation of each test case to its K (which can be set in the algorithm) nearest neighbors (vector representations of test cases). The overall procedure of this technique can be found in Algorithm 2.

Algorithm 2 Average Minimum Distance (AMD) algorithm

Data: Dataframe of (Test Name, Embedding Vector)
FT_Name: Name of Failed Test
K: Number of nearest neighbours Failed test's normalized rank

```
procedure ComputeRank(Data, FT_Name, K)
    Test_Scores ← Empty Dictionary
    for i, trace in enumerate(Data) do
        distances ← cosine_distances(Data.vectors , Data.vectors[i])
        distances ← sort(distances)
        top_k ← distances[1 : K + 1]
        Test_Scores[trace.name] ← mean(top_k)
    end for
    Test_Scores.sortValues(by =′ values′, Descending = True)
    Rank ← Test_Scores.keys().index(FT_Name)
    Normalize_Rank ← normalize(Rank, Data.length)
    return Normalize_Rank
end procedure
```

COV measure: In order to prioritize the test cases using this technique, we start by choosing a random test case. After choosing the first test case, we add the next test case that maximizes the COV value of the selected test cases. Equation 5.3 shows the formula

for calculating the COV value for a set of N points ($\nu = \{z_i\}_{i=1}^{N}$).

$$\begin{aligned} \gamma_i &= min_{j \neq i} |z_j - z_i| \\ COV(\nu) = \lambda &= \frac{1}{\overline{\gamma}} \sqrt{\left(\frac{1}{N} \sum_{i=1}^{N} (\gamma_i - \overline{\gamma})^2 \right)} \end{aligned} \tag{5.3}$$

We repeat this step until we find the failing test rank. The overall procedure of this technique can be found in Algorithm 3.

Algorithm 3 COV measure algorithm

Data: Dataframe of (Test Name, Embedding Vector)
FT_Name: Name of Failed Test Failed test's normalized rank

```
1: procedure ComputeRank(Data, FT_Name, K)
2:     Selected_Suite ← Empty List
3:     initial_testcase ← Data.pop(random(Data.keys()))
4:     Rank ← 1
5:     while FT_Name in Data.keys() do
6:         test_score ← Empty Map(TestName, COVscore)
7:         for item in Data do
8:             tmp_list ← Selected_suite.copy()
9:             tmp_list.append(item.value)
10:            Test_score['item.key'] ← COV(tmp_list)
11:        end for
12:        Selected_test ← max(test_scores, key = test_score.get)
13:        Data.pop(selected_test)
14:        Rank ← Rank + 1
15:    end while
16:    Normalize_Rank ← normalize(Rank, Data.length)
17:    return Normalize_Rank
18: end procedure
```

In **RQ3-2**, we compare the best diversification-based TP model from the last research question (Average Minimum Distance) with the classifier-based TP model that is proposed in the first research question to see which approach works better for TP.

In **RQ3-3**, we design the combined-TP approach. Recall that our hypothesis is that when the failing test case is similar to the historical failures the classifier-TP usually works better. Whereas, when the failing test is not close to historical failures, and it is more like an anomaly in the test suite, the diversification-TP works better. Therefore, the proposed

combined-TP approach merges the two heuristics, by deciding which one to use per test suite. It uses a logistic regression model as a classifier to decide which technique should be used for detecting the failing test case. This classifier uses two features for prediction. The first feature is the average probability calculated by the softmax model for the top 5 test cases in the ranking. The second feature is the average distance of the top 5 most outlying points to the center of the test suite divided by the average distance of all points to the center.

Intuitively speaking, the combined-TP approach checks the top N tests ranked by each heuristic and chooses the heuristic which is more confident in its ranking predictions (either through probabilities or distances). The reason to use the probabilities and distances of top N tests rather than all test cases is that there are only a few failing tests (mostly just one) per test suite. Therefore, the ranks of the rest of the test cases are not predictive of the effectiveness of the classifier-TP vs. diversification-TP and act more like noise in the combined-TP's decision. The actual value of N is a hyper-parameter for classifier-TP and N=4 was chosen here based on our small tuning experiments with N= 1 to 10. In terms of evaluating the TPs in each RQ, we have different approaches for RQ1 and 2. In RQ1, we use both FFR and APFD. Regarding the FFR analysis, all the coverage-based, classifier, and diversification TPs are applied to the failed version of each revision. To have a statistically and practically meaningful TP analysis using our metrics, we have only included those failing test suites, from the test dataset, that have more than 5 test cases, per buggy method. This results in excluding 5 out of 30 failures (one version from Cli, one version from Compress, and three versions from Mokito).

Regarding the APFD analysis, the traces are collected from the mutants as explained in 5.2) so there is no concern about the limited number of data points, as in the FFR analysis.

In RQ3, we only use FFR. The reason is that diversification-TP works based on detecting a few outliers among mostly typical test cases in a test suite. That's why if we seed hundreds

of fake faults, by mutation to measure APFD, the bugs can not all be correctly identified as anomalies.

In all three RQs, the TPs that have a random component (coverage-based-TP techniques, since there are many test cases with the same coverage score, where we had to break the tie randomly) are executed 30 times per version and we record the median values (AFPD and FFR) of those 30 runs for that version. To compare two TPs in any RQ, we look at the mean, median, and statistical significant test results of FFRs and APFDs over the 30 (5X6) project versions. Therefore, depending on the TP, each of these 50 values per distribution is either an individual metric (AFPD or FFR) or a median of 30 measurement of that metric.

We use a paired non-parametric statistical significance test (Wilcoxon signed rank test [73] with a p-value less than 0.05) and calculated the effect size (rank-biserial correlation [73]), every time we compare two distributions of ranks over their 30 samples.

5.5 Configurations & Environments

We compile and run projects with *Java1.7*, as required by the versions in Defects4J. Our code is implemented in *Python3.9.7*. For model training, we have used *TensorFlow 2.7.0* and *Keras* libraries for implementing our neural network model. We, also, have used the CodeBERT implementation from the *huggingface/transformers* framework and loaded the pre-trained CodeBERT-base weights. The test vector size (i.e., number of dimensions in the latent space) is set to 100 where each vector generated from each CodeBERT model has the size of 768. This configuration was found after some ad-hoc tuning but it might not be optimal and it can be easily set to any size in the configuration file. Also, the models were trained to 40 epochs, and each CodeBERT model was fine-tuned for 10 epochs. Training and evaluation of the deep learning model were done on a single node running Ubuntu 18.04, using 32 CPUs, 250G memory, and a Tesla V100 GPU.

The time that we consumed for fine-tuning the CodeBERT model varied depending on the

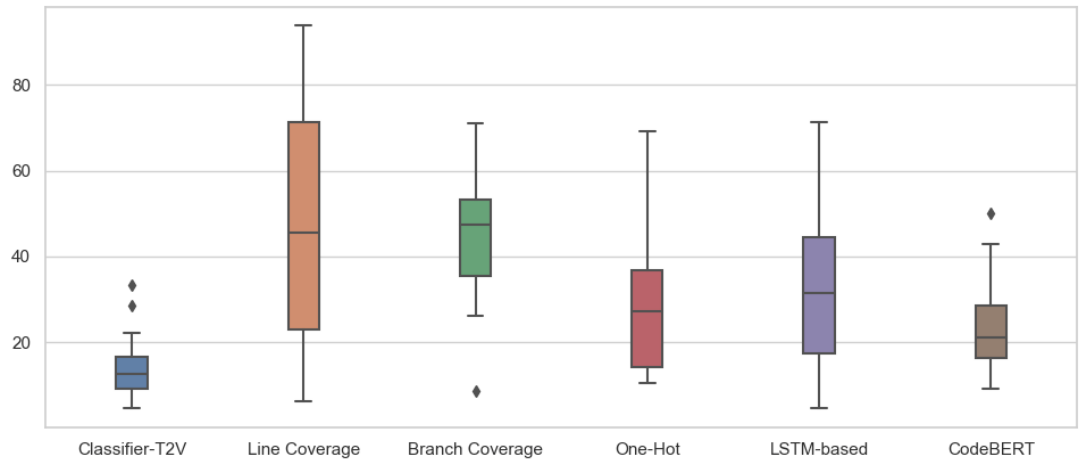


Figure 5.3: Boxplots of the normalized ranks for the first failing test case for RQ1, prioritized by each TP technique.

size of the project. For example, it took about 26 hours to fine-tune the CodeBERT model in project Lang. In comparison, the training model for Test2Vec was around 37 hours. Note that training jobs are not frequent (once per project) and the inference times (vectorizing the test suite of a version based on the trained model) are negligible (a few seconds). Finally, prioritization time differences between different TP methods are also practically negligible (on average less than a second) and literally the same for all diversification and classifier-TP techniques.

5.6 Results

In this section, we report and discuss the results of our experiments, per RQ.

RQ1) How effective is Test2Vec embedding for test prioritization when the tests are ranked using a classifier trained on historical data?

To address this RQ, we compared Test2Vec classifier-TP with baselines in two categories: basic traditional TPs (RQ1-1) and alternative embedding (RQ1-2).

RQ1-1) Comparing with Coverage-based TPs:

As shown in Table 5.2, among all 25 (30 - the 5 versions that did not have enough tests in the failing test suite) faulty versions under study, the normalized rank of the first failing

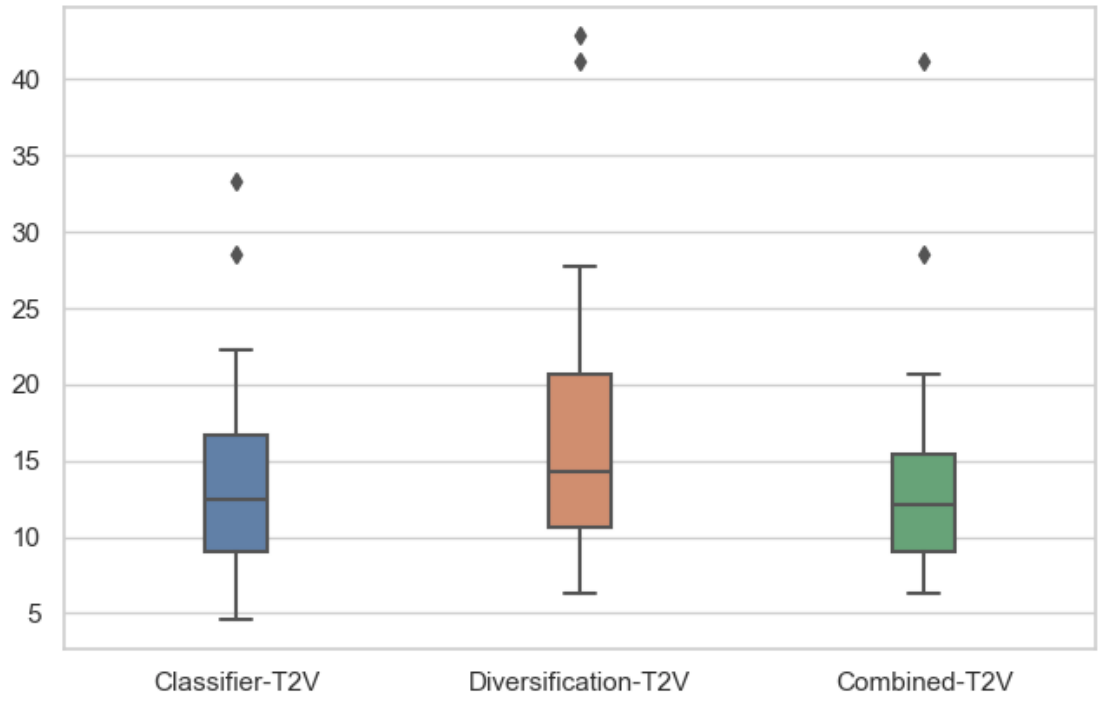


Figure 5.4: Boxplots of the normalized ranks for the first failing test case for RQ3, prioritized by each TP technique.

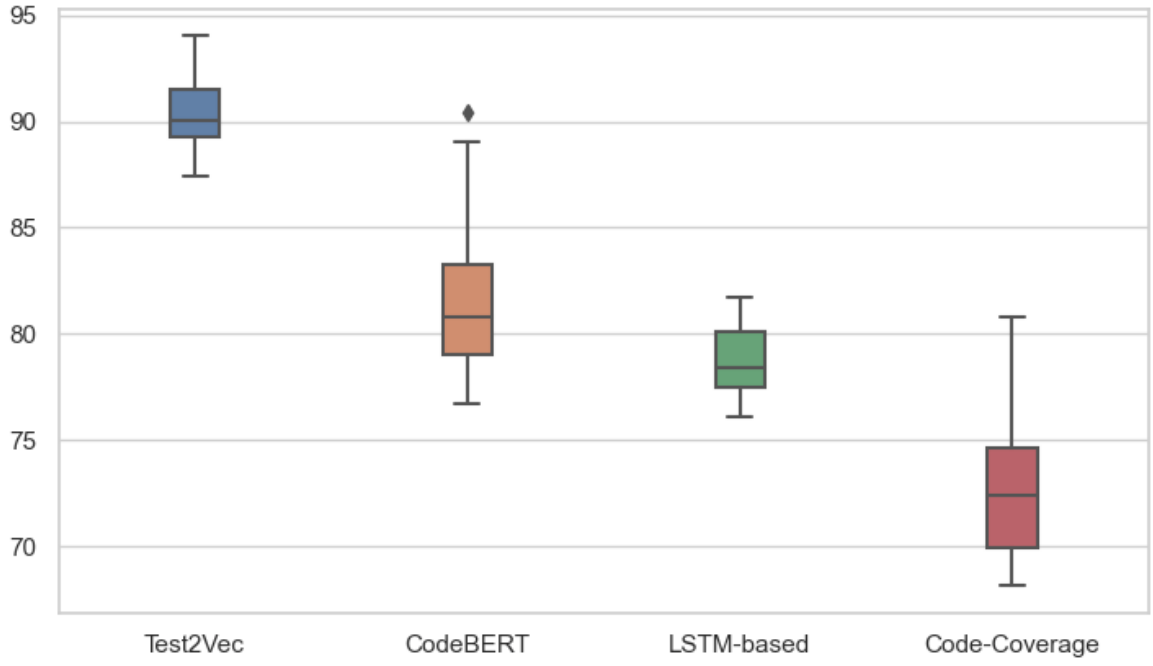


Figure 5.5: Boxplots of the APFD, prioritized by each TP technique.

test (FFR) in the original faulty versions for the Test2Vec classifier-TP has an average of 14.01 and a median of 12.50. In comparison, the line coverage(LC)-based TP has an average of 46.82 and a median of 45.61, and the branch coverage(BC)-based TP technique has an average of 46.36 and a median of 47.29. The relative improvement of the Test2Vec over the coverage-based TPs are: 70.93% (average) and 72.59% (median) over LC-based TP; and 69.79% (average) and 73.56% (median) over BC-based TP.

Table 5.2: Statistics on the normalized ranks (FFRs) for RQ1-1, over all 25 versions under study, grouped by projects. The FFRs of coverage based TPs per version are the median of 30 runs. All median and AVG data are across the versions per project. The bold values are the dominating results.

	Test2Vec		Line Coverage		Branch Coverage	
	Median	AVG	Median	AVG	Median	AVG
Cli	**12.86**	**13.03**	36.37	41.89	44.65	43.14
Jsoup	**12.12**	**14.89**	18.18	21.89	46.81	44.94
Compress	**15.96**	**16.24**	69.04	54.53	48.82	49.21
Lang	**9.09**	**10.46**	45.61	57.22	50.90	47.34
Mokito	**18.25**	**18.25**	73.00	73.00	25.78	25.78
Math	**11.11**	**14.07**	48.125	48.66	47.29	47.09
Total	**12.50**	**14.01**	45.61	46.82	47.29	46.36

Tabel 5.3 also shows the APFD for all execution traces (including failing traces generated by mutation) among all 25 versions under study. the APFD for the Test2Vec classifier-TP has an average of 90.19 and a median of 90.20. In comparison, the line coverage(LC)-based TP has an average of 71.86 and a median of 71.51, and the branch coverage(BC)-based TP technique has an average of 70.01 and a median of 70.22. The relative improvement of the Test2Vec over the coverage-based TPs are: 20.33% (average) and 20.72% (median) over LC-based TP; and 22.38% (average) and 22.14% (median) over BC-based TP.

Figures 5.3, and 5.5 report the whole distributions, as boxplots, of the normalized rank and the APFD, after repeating the experiment 30 times (by selecting randomly from test cases with a tied score) for all versions in every project, as boxplots. As the figure shows, Test2Vec classifier-TP not only offers better median and average test case ranking but also

Table 5.3: Statistics on the APFDs, over all 25 versions under study, grouped by projects for RQ1-1. The APFDs of coverage based TPs per version are the median of 30 runs. All median and AVG data are across the 5 versions per project. The bold values are the best results.

	Test2Vec		Line Coverage		Branch Coverage	
	Median	AVG	Median	AVG	Median	AVG
Cli	**94.32**	**92.20**	71.79	73.89	69.17	68.22
Jsoup	**91.11**	**90.37**	68.96	70.83	70.54	69.92
Compress	**90.03**	**89.43**	75.01	74.21	69.67	70.61
Lang	**88.51**	**87.94**	72.31	72.18	71.89	72.79
Mokito	**89.81**	**90.14**	69.58	69.09	69.95	70.53
Math	**90.73**	**91.21**	69.47	70.14	68.74	68.06
Total	**90.20**	**90.19**	71.51	71.86	70.22	70.01

is more reliable, given the smaller variation over the randomized runs and versions.

Running the statistical tests (Wilcoxon Signed-Rank) on the FFR results shows that the effect size and p-value when we are comparing the Test2Vec classifier-TP to the LC-TP, are 0.85 and 1.1e-6. When we Compare Test2Vec classifier-TP and the BC-TP results, the effect size is 0.87 and the p-value is 5.9e-8.

Table 5.2 also shows that, surprisingly, LC-based TP outperforms BC-based TP. This seems to contradict the fact that BC subsumes statement coverage. However, coverage subsumption does not necessarily translate into a better TP. This might simply be due to extra information (noise) in the BC that is harming the TP rather than being beneficial.

It also indicates that the representation itself is important; simply adding richer information (BC vs LC) does not guarantee improved performance.

RQ1-2) Comparing with state-of-art code and execution trace embeddings:

Table 5.4 shows the FFR for the one-hot, CodeBERT, and LSTM-based TP methods. The one-hot TP method has an average of 29.97 and a median of 27.27, the CodeBERT TP technique has an average of 23.52 and a median of 21.05, and the LSTM-based method has an average of 33.27 and a median of 31.57. The relative improvement of the Test2Vec over the encoding and embedding baseline TPs are: 53.28% (average) and 54.16% (median)

over One-Hot TP; 40.47% (average) and 40.62% (median) over CodeBERT TP; and 57.92% (average) and 60.41% (median) over LSTM-based TP.

Table 5.4: Statistics on the normalized ranks (FFRs) for RQ1-2, over all 25 versions under study, grouped by projects. The FFRs of coverage based TPs per version are the median of 30 runs. All median and AVG data are across the versions per project. The bold values are the dominating results.

	Test2Vec		One-Hot		LSTM-based		CodeBERT	
	Median	AVG	Median	AVG	Median	AVG	Median	AVG
Cli	**12.86**	**13.03**	29.79	30.04	34.65	30.58	17.42	18.05
Jsoup	**12.12**	**14.89**	27.27	33.68	24.13	35.27	30.43	28.50
Compress	**15.96**	**16.24**	15.96	21.82	34.82	38.20	20.53	23.92
Lang	**9.09**	**10.46**	27.28	35.41	31.57	27.61	18.18	22.39
Mokito	**18.25**	**18.25**	26.98	26.98	36.50	36.50	25.39	25.39
Math	**90.73**	**91.21**	31.25	28.50	28.57	33.87	22.22	22.99
Total	**12.50**	**14.01**	27.27	29.97	31.57	33.27	21.05	23.52

For the APFD experiment, as Table 5.5 suggests, the APFD for the one-hot TP method has an average of 77.76 and a median of 79.14, the CodeBERT TP technique has an average of 88.40 and a median of 88.13, and the LSTM-based method has an average of 77.51 and a median of 77.72. This means that the relative improvement of the Test2Vec over the encoding and embedding baseline TPs are: 13.79% (average) and 12.25% (median) over One-Hot TP; 1.99% (average) and 2.28% (median) over CodeBERT TP; and 14.07% (average) and 13.83% (median) over LSTM-based TP.

Statistical tests results when we are comparing the Test2Vec classifier-TP with the One-Hot, CodeBERT, and LSTM-based one are effect size 0.76 and p-value 2.5e-4; effect size 0.79 and p-value 3.8e-4; and effect size 0.76 and p-value 6.0e-5, respectively.

The results show that (a) a simple encoding such as One-Hot cannot capture much relevant information from traces, for the TP task and that (b) CodeBert outperforms the LSTM-based embedding. This is interesting since the LSTM-based approach is specialized for execution traces and testing but it can not compete with off-the-shelf CodeBert. Furthermore, the results show that (c) our proposed embedding outperforms alternatives. This is notable, in

Table 5.5: Statistics on the APFDs, over all 25 versions under study, grouped by projects for RQ1-2. The APFDs of coverage based TPs per version are the median of 30 runs. All median and AVG data are across the 5 versions per project. The bold values are the best results.

	Test2Vec		One-Hot		LSTM-based		CodeBERT	
	Median	AVG	Median	AVG	Median	AVG	Median	AVG
Cli	**94.32**	**92.20**	76.23	78.93	81.19	78.21	88.31	87.74
Jsoup	**91.11**	**90.37**	75.93	74.17	76.93	77.23	90.17	90.44
Compress	**90.03**	**89.43**	77.30	79.36	81.27	81.09	94.43	88.53
Lang	**88.51**	**87.94**	76.41	75.49	72.94	73.62	82.19	86.49
Mokito	**89.81**	**90.14**	81.87	80.01	80.11	80.15	82.58	84.72
Math	**92.00**	**91.42**	79.52	80.52	74.31	77.19	91.01	90.17
Total	**90.20**	**90.19**	79.14	77.76	77.72	77.51	88.13	88.40

Table 5.6: Statistics for RQ1, comparing with Test2Vec classifier-TP

	FFR	
	P-Value	Effect-Size
Line Coverage	1.1e-6	0.85
Branch Coverage	5.9e-8	0.87
One-Hot	2.5e-4	0.76
LSTM-based	3.8e-4	0.79
CodeBERT	6.08e-5	0.76

particular since the advanced embedding models (CodeBERT and LSTM) use the same TP algorithms as Test2Vec (they all use the same pooling and softmax layers for calculating the ranks), and they are trained, validated, and tested on the same dataset splits. Thus the differences in the final results are due to the architecture of our proposed embedding.

Taken together, the RQ1 results show that the default Tes2Vec setup (Test2Vec classifier-TP) is significantly better than the coverage-based TP and state-of-art embedding TP techniques we compared it to. Comparing Test2vec classifier-TP with the best coverage-based (LC) and the best embedding (CodeBERT), the relative improvement regarding FFR (at the test suite level) are 70.93% (average) and 72.59% (median) over LC; and 40.47% (average) and 40.62% (median) over CodeBERT. Furthermore, in terms of APFD (in the whole project-version level), the relative improvements are 20.33% (average) and 20.72% (median) over LC; and 1.99% (average) and 2.28% (median) over CodeBERT.

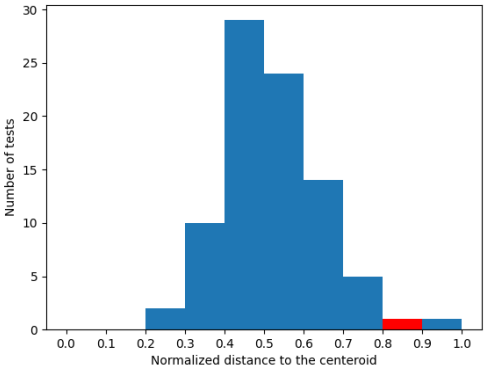
RQ2)Does the Test2Vec embedding benefit from input/output parameter values in the execution traces, to better prioritize test cases?

As discussed in section 5.4, in order to see if our model benefits from input/output sequences, we compared three models together: Test2Vec classifier-TP, Test2Vec(no-output) classifier-TP, and Test2Vec(no-I/O) classifier-TP. Table 5.7 shows the median (average) FFR results for these models. The Test2Vec(no-output) classifier-TP method has an average of 17.76 and a median of 15.38, and the Test2Vec(no-I/O) classifier-TP technique has an average of 26.21 and a median of 26.31. The relative improvement of the Test2Vec over the encoding and embedding baseline TPs are: 21.16% (average) and 18.75% (median) over Test2Vec(no-output) classifier-TP; and 46.57% (average) and 52.50% (median) over Test2Vec(no-I/O) classifier-TP.

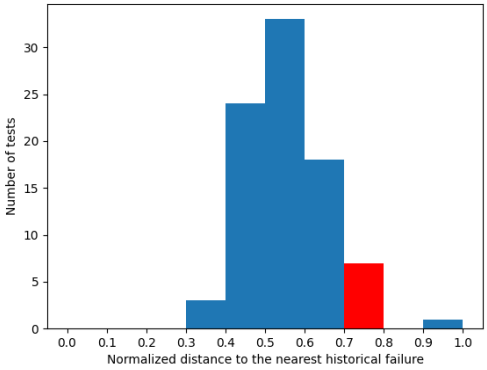
Statistical tests results when we are comparing the Test2Vec classifier-TP with the Test2Vec(no-output) classifier-TP and Test2Vec(no-I/O) classifier-TP one are effect size 0.74 and p-value 1.4e-2; and effect size 0.87 and p-value 4.8e-4, respectively.

The results show that (a) Only leveraging the method call sequences cannot capture much relevant information from traces (b) Using input sequence alongside the method call sequence can significantly improve the ability of the model for the task of TP (c) using both input

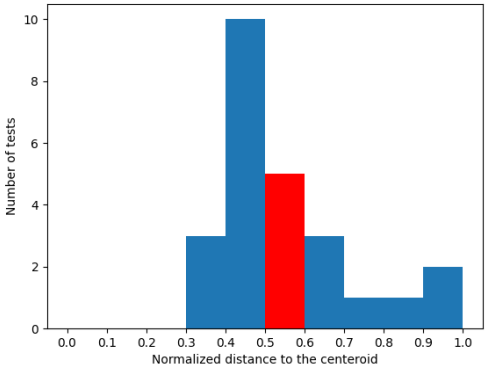
Figure 5.6: Illustrated distances of two sample test cases from the centroids of their test suites and their projects' historical failing tests, to motivate the Combined-TP approach. The four sub-figures show the distribution of the normalized distance of each trace from: their centroids in the latent space of their current test suite (a and c) and their nearest trace within the historical failing traces (b and d). The closer values to 1.0 in sub-figures a and c means the more anomalous behaviour. The closer value to 0.0 in sub-figures b and d means the closer behavior to past failing traces. Sub-figures (a) and (b) are from version 170 of the Chart project and sub-figures (c) and (d) are from the version 33 of Cli project.



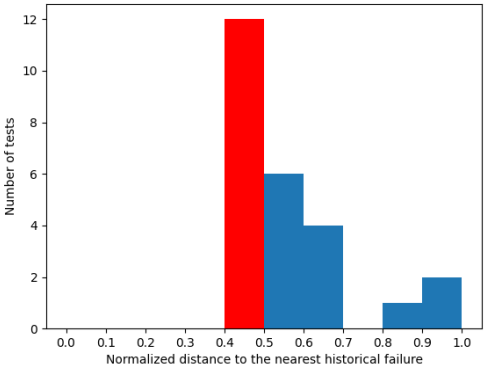
(a)



(b)



(c)



(d)

Table 5.7: Statistics on the normalized ranks (FFRs) for RQ2, over all 25 versions under study, grouped by projects. The FFRs of coverage based TPs per version are the median of 30 runs. All median and AVG data are across the versions per project. The bold values are the dominating results.

	Test2Vec		Test2Vec (no-output)		Test2Vec (no-I/O)	
	Median	AVG	Median	AVG	Median	AVG
Cli	**12.86**	**13.03**	14.34	13.92	20.46	21.99
Jsoup	**12.12**	**14.89**	17.33	13.79	27.78	27.27
Compress	**15.96**	**16.24**	21.82	23.10	27.98	27.38
Lang	**9.09**	**10.46**	15.71	15.38	24.90	19.17
Mokito	**18.25**	**18.25**	18.25	18.25	38.09	38.09
Math	**11.11**	**14.07**	19.52	12.50	24.37	27.27
Total	**12.50**	**14.01**	15.38	17.76	26.31	26.21

and output sequences with method call sequences can lead to the embeddings that capture most relevant information for test case executions in the task of TP.

Taken together, the RQ2 results show that the default Test2Vec classifier-TP outperforms the Test2Vec(no-output) TP and Test2Vec(no-I/O) TP. Comparing Test2vec classifier-TP with Test2Vec(no-output) classifier-TP and Test2Vec(no-I/O) classifier-TP, the relative improvements regarding FFR (at the test suite level) are 21.16% (average) and 18.75% (median); and 46.57% (average) and 52.50% (median), respectively.

RQ3) Which of the two prioritization heuristics (similarity to past failing tests or test diversification) is better to be used with Test2Vec for a test case prioritization task?

To see if we can leverage the diversification for test case prioritization, we analyzed the results of the following sub-RQs:

RQ3-1) Comparing diversification techniques: Table 5.8 shows the normalized FFR among all 25 (30 - the 5 versions that did not have enough tests in the failing test suite) faulty versions under study for three different diversification-based TP algorithms. The

Figure 5.7: Comparing the normalized rank of first failing test (FFR) for diversification- and classifier-TP.

normalized rank of the first failing test (FFR) in the original faulty versions for the Test2Vec diversification(AMD)-TP has an average of 17.44 and a median of 14.28. In comparison, the Test2Vec diversification(Distance-to-Centroid)-TP has an average of 20.01 and a median of 15.06, and the Test2Vec diversification(COV)-TP technique has an average of 21.26 and a median of 21.76. The relative improvement of the Test2Vec diversification(AMD)-TP over the other diversification-based TPs are: 12.84% (average) and 5.19% (median) over diversification(Distance-to-Centroid)-TP; and 17.98% (average) and 34.36% (median) over Test2Vec diversification(COV)-TP.

Overall, the local anomaly detection algorithm (Average Minimum Distance) shows better results in comparison with other diversification-based TP algorithms. Therefore, we have chosen this technique as our default diversification-based TP algorithm for the rest of this research question.

RQ3-2) Comparing diversification with classifier-TP:

As discussed in section 5.4, diversification-TP is only evaluated using real faults (by reporting FFRs) and not mutants (i.e., no APFD). Table 5.9 shows the median (average) FFR results for both the classifier- and diversification-TPs.

Table 5.8: Statistics on the normalized ranks (FFRs) for RQ3-1, over all 25 versions under study, grouped by projects. The FFRs of coverage based TPs per version are the median of 30 runs. All median and AVG data are across the versions per project. The bold values are the dominating results.

	AMD		Distance-to-Centroid		COV measure	
	Median	AVG	Median	AVG	Median	AVG
Cli	**15.15**	**17.75**	17.03	18.58	21.99	23.21
Jsoup	**20.68**	**19.39**	20.68	23.33	21.73	26.66
Compress	**16.51**	**22.12**	14.28	21.94	33.03	33.11
Lang	**10.52**	**12.93**	13.63	16.65	15.78	15.79
Mokito	**12.69**	**12.69**	12.69	12.69	25.39	25.39
Math	**9.09**	**17.90**	18.18	22.57	22.22	26.21
Total	**14.28**	**17.44**	15.06	20.01	21.76	21.26

The median FFRs for the Test2Vec diversification-TP method has an average of 17.44 and a median of 14.28. This means that the classifier-TP technique is dominating by a relative improvement of 19.71% (average) and 12.50% (median).

However, as shown in Figure 5.7, in some cases, diversification-TP can predict the failing test case in a better rank than classifier-TP. We can justify this with the fact that historical failures do not contain all the failing behaviors. Therefore, these failing behaviors may be detected as an anomaly within the latent space rather than by comparing them to historical failures. In other words, and for the specific projects we investigated, depending on the type of fault, about 84% of the time the similarity to past failures is a better heuristic, whereas about for 16 % of the failures, detecting anomalous behavior within a test suite is a better heuristic. Thus, for other projects and circumstances, such as the length of the history, the strength/maturity of the test suite, etc., one or the other of these heuristics might be preferable.

To give an example of how different faults can be better detected by one technique than the other, let's look at two sample failing tests from our dataset, illustrated in figure 5.6. The sub-figures show two statistics per sample: (a) how anomalous the failing test case is within its test suite (measured by its vector's distance to the centroid of the failing test suite

Table 5.9: Statistics on the normalized ranks (FFRs) for RQ3-2 and RQ3-3, over all 25 versions under study, grouped by projects. The FFRs of coverage based TPs per version are the median of 30 runs. All median and AVG data are across the versions per project. The bold values are the dominating results.

	Classifier-TP		Diversification-TP		combined-TP	
	Median	AVG	Median	AVG	Median	AVG
Cli	**12.86**	**13.03**	15.15	17.75	10.79	11.62
Jsoup	**12.12**	**14.89**	20.68	19.39	13.04	14.32
Compress	**15.96**	**16.24**	16.51	22.12	14.28	18.99
Lang	**9.09**	**10.46**	10.52	12.93	9.09	10.18
Mokito	**18.25**	**18.25**	12.69	12.69	12.69	12.69
Math	**11.11**	**14.07**	9.09	17.90	11.11	14.64
Total	**12.50**	**14.01**	14.28	17.44	12.12	13.74

– the more the better anomaly), and (b) how similar it is to the failing tests in the past (measured by its distance to the closest failing trace from historical versions – the less the closer to past failures). In figure 5.6.b, we see that for version 170 of the Chart project, using diversity to detect the failing test is a better heuristic, since the failing test's trace is not very similar to historical failing traces (the normalized distance to the closest failure is between 0.7 and 0.8, figure 5.6.b), but it looks quite like an anomaly in the current test suite (the normalized distance from the centroid is between 0.8 and 0.9, figure 5.6.a). On the other hand, a failing test case such as the one in version 33 of the Cli project, can be better detected by the classifier-TP, since the failing test's trace is quite close to at least one historical failure (the normalized distance to the closest failure is between 0.4 and 0.5, figure 5.6.d), but does not look like an anomaly within the test suite (the normalized distance from the centroid is between 0.5 and 0.6, figure 5.6.c)

Based on the above statistics and the provided example, given that the two types of failures are not necessarily overlapping, techniques that combines the two heuristics, such as the combined approach we evaluated in RQ3-3 might provide benefits.

RQ3-3) Combining classifier- and diversification-TPs:

As discussed, although the default Test2Vec classifier-TP outperforms the Test2Vec diversification-

TP method, overall, there seem to be non-overlapping cases that the diversification heuristic is a better approach for detecting failing behaviors and thus test prioritization. In this sub-RQ, we compare the Test2Vec combined-TP with the default classifier-TP. As shown in Table 5.6, the average and median of FFR for the combined-TP method is 13.74 and 12.12. This means the combined-TP method slightly outperforms the classifier method by 1.84% (average) and 3.03% (median). However, Table 5.10 shows that the differences between the two techniques are not statistically significant.

Table 5.10: Statistics on the normalized ranks (FFRs) for RQ3, over all 25 versions under study (the high effect size indicates the left-side technique, per row, is outperforming the right-side technique).

	FFR	
	P-Value	Effect-Size
Classifier-TP vs Diversification-TP	0.0048	0.75
Combined-TP vs Diversification-TP	0.0015	0.87
Combined-TP vs Classifier-TP	0.76	0.56

The more interesting observation is perhaps the fact that Test2Vec combined-TP's median FFR, over 5 versions of each project, is better or the same as Test2Vec classifier-TP's results in 4 out of 6 projects. Looking into all 25 versions together, in 21 versions Test2Vec combined-TPs' FFR is better or equal to Test2Vec classifier-TP's FFR.

We also observe that the classifier used in the combined-TP method for selecting between the classifier- and diversification-TP can successfully detect the correct test case prioritization method in 84% of cases. Still, the simple selection method we have used can likely be improved upon; we leave it to future work to further investigate this.

Comparing our best proposal (Test2Vec combined-TP) with the best coverage-based and best alternative embedding from RQ1 also shows significant relative improvements. 70.64% and 41.57% in terms of FFR, over the best coverage and best alternative embedding, respectively.

Overall, although the improvements of Test2Vec combined-TP over Test2Vec classifier-TP are not significant, given that in 84% of the cases Test2Vec combined-TP is better or

the same as the default version, we recommend it over the classifier-TP.

> Take together, RQ3 results show that the default Tes2Vec setup (Test2Vec classifier-TP) is significantly better than all alternatives including Test2Vec diversification-TP. However, the two heuristics for Test2Vec (i.e., similarity to failing tests from previous versions and being an anomaly in the current test suite) are complementary and the combined approach is slightly better than the default version.

5.7 Discussion on limitations of the proposed approach and the conducted study

The first limitation of our approach is that we need a training set of execution traces. Depending on the testing task and context, this may not be available. For instance, to use Test2Vec in test generation, we have to either start with an existing developer-written test suite or use an existing automated test generation tool to build one. Then Test2Vec can learn the representation of existing tests, before guiding the generation of new ones.

Another limitation is that, even with enough tests in the training set, if the number of real bugs are limited (which they usually are) we have to use mutation to create a (more) balanced dataset of fail/pass traces. But generating mutants and collecting traces for buggy tests in the mutated versions is time consuming. However, this is NOT a frequent task; in many cases it should be enough to do this once per project. As long as there aren't many changes to the project, the trained model can be reused. Therefore, we do not need to mutate it or collect traces, per change. The cost both of mutation and of creating embeddings can thus likely be amortized over many versions of a project, reducing the per-version cost. Future work is needed to investigate this and establish the cost versus performance trade-offs involved.

The training time for our approach is not negligible and can be substantial.Even if, as

discussed above, this can be amortized over several versions, it is a limitation that over time, and multiple versions, the model's accuracy may decline.

Generally speaking, balancing the trade-off between the cost of re-training and still having an accurate model without re-training is a problem we did not address in this thesis. In our experiments, we looked at the results with one-time training (on 20 versions) and five times reuse. We did not observe a significant degradation over those five uses. But a more comprehensive study could investigate when and how to re-train the models. It is also worth mentioning that retraining may affect both the accuracy of the embeddings as well as the downstream task procedure. In our experiments, the effect of retraining can be bolder on the classifier-TP compared to diversification-TP, given that classifier-TP's accuracy partly depends on how rich and relevant the historical failures still are. However, Diversification-TP is only affected in the embedding generation phase and the ranking procedure only depends on the current test suite.

One limitation of the experiment (not the approach itself) is we implemented the TPs using a greedy approach (same across the proposed approaches and baselines) for optimizing the rank list. More recent TP papers try to use optimization algorithms such as evolutionary algorithms [55]. Given that our focus was on the test representation and not the optimization approach, we did not explore this dimension. In other words, we argue that the representation and the optimization based on it are orthogonal choices and this thesis only focuses on the former. Future work should investigate if optimization on top of the proposed embeddings can provide additional benefits. It would be beneficial to explore further many of the design choices in this study. For instance, one study can be designed to compare the effect of different diversity functions on Test2Vec diversification-TP results. Another study can be on exploring the idea of input parameter abstraction in more detail to come up with different alternatives to compare.

Another limitation of the experiment was that we applied our approach at the unit testing level by eliminating nested calls. By focusing on inter-class interaction, future works can

apply this same approach to integration testing as well.

Finally, we used the CodeBERT for token embedding generation for outputs, method calls, and input sequences. This means that we have a 512 tokens limitation for each sequence. There are many works on the domain of NLP for handling larger sequuences [72, 10]. The same ideas can be applied to our problem in future works.

5.8 Validity threats

Regarding construct validity, in projects like Lang and Time, the traces' length varies from a few calls to thousands of calls. Therefore, in the pre-processing step, we perform padding or truncation to get a fixed-length set of traces. This may cause information loss in very long traces, i.e., our representation of those test cases does not accurately measure what it claims (its behavior). However, these cases are rare and likely would not change the overall findings.

In terms of internal validity, we chose our evaluation metrics quite carefully! We wanted metrics that work on both real-world bugs as well as mutations. We also wanted to be able to evaluate our TPs both on the whole project version as well as the test suite level. The rank of first failing test is suitable for real-world faults, where we have only one fault in the class under test with very few failing test cases in the failing suite. But we had to normalize it since without normalization large and small test sets' results were not comparable. The APFD, on the other hand, is used for the TP task when we use mutation analysis on the whole version's test cases, where we have multiple mutants (bugs) per class, resulting in multiple failing tests. Although the caveat of APFD was that we could not use APFD in RQ3 (since diversification in the presence of multiple mutants would not give any meaningful results; so many anomalies won't make sense), at least for RQ1 we managed to explore the results from all aspects (real-world vs mutants AND test suite vs whole version test cases).

In terms of conclusion validity, we repeat our training on 6 different projects and report

the statistical tests and effect size across multiple versions. We also repeat the TPs for 30 runs, whenever the TP had randomness in the process, per version. We then carefully reported both median and mean, as well as paired non-parametric statistical tests and effect sizes, to better analyze the differences and their statistical and practical significance.

Finally, in terms of external validity, although we tried to mitigate the threat by selecting multiple (6) projects from a benchmark dataset, our findings are limited to TP in the context of unit testing of Defect4J projects. So our findings can not be generalized for other types of test cases or other projects especially industrial ones, without further replications.

Chapter 6

Conclusion and Future Works

In this thesis, we proposed a novel neural embedding model called Transformer Test2Vec, that embeds test cases as numerical vectors which can, in turn, improve performance on downstream testing tasks such as test prioritization. The flexibility of neural embeddings enables the representation of rich test case information such as the ordering of method calls as well as specific test input and output values.

Empirical evaluation, of six projects from Defects4J, shows that traditional code coverage metrics (Line Coverage and Branch Coverage), which are state-of-practice for many test-related tasks, are weak representations for test cases regarding identifying the failing behaviors. It also shows that our approach outperforms traditional code coverage metrics, classic diversity-based encoding, state-of-the-art neural program embedding models, and embedding with other sequence learning approaches, on a test prioritization task.

We also compared two different heuristics for identifying the failing behavior in a test suite using embeddings generated with our model, namely similarity to the historical failures and diversification. Results show that similarity to the historical failure is significantly better in the task of test case prioritization. However, further analyses showed that using diversification can identify certain types of failures better than the similarity-based approach. Therefore, we proposed an approach that leverages both techniques in order to identify

failing behaviors (combined method). Results show that the combined method can slightly outperform the similarity-based approach.

Future work should investigate the effect of including, even more, relevant information in the execution traces as well as study the use of Test2Vec embeddings for other downstream testing tasks such as automated test generation. We will also investigate the applicability of this approach, by using other dynamic representations of the program behavior, on the system, security, and performance testing. Newer transformer models, such as CodeX [15] and GPT-3 [13], have recently been introduced. The impact of using more advanced Transformer-based models on the TP also needs to be investigated in future works.

Bibliography

[1] S. Ali, L. C. Briand, H. Hemmati, and R. K. Panesar-Walawege. A systematic review of the application and empirical investigation of search-based test case generation. *IEEE Transactions on Software Engineering*, 36(6):742–762, 2010.

[2] Shaukat Ali and Hadi Hemmati. Model-based testing of video conferencing systems: Challenges, lessons learnt, and results. In *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation*, pages 353–362, 2014.

[3] Miltiadis Allamanis, Earl T. Barr, Premkumar Devanbu, and Charles Sutton. A survey of machine learning for big code and naturalness. *ACM Comput. Surv.*, 51(4), July 2018.

[4] Felipe Almeida and Geraldo Xexéo. Word embeddings: A survey. *CoRR*, abs/1901.09069, 2019.

[5] Uri Alon, Shaked Brody, Omer Levy, and Eran Yahav. code2seq: Generating sequences from structured representations of code, 2019.

[6] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. code2vec: Learning distributed representations of code, 2018.

[7] David Azcona, Ihan Hsiao, Piyush Arora, and Alan Smeaton. User2code2vec: Embeddings for profiling students based on distributional representations of source code.

In *Proceedings of the 9th International Conference on Learning Analytics and Knowledge*, ACM International Conference Proceeding Series, pages 86–95. Association for Computing Machinery, March 2019.

[8] Earl Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. The oracle problem in software testing: A survey. *IEEE Transactions on Software Engineering*, 41:1–1, 01 2014.

[9] Iz Beltagy, Arman Cohan, and Kyle Lo. Scibert: Pretrained contextualized embeddings for scientific text. *CoRR*, abs/1903.10676, 2019.

[10] Iz Beltagy, Matthew E. Peters, and Arman Cohan. Longformer: The long-document transformer. *CoRR*, abs/2004.05150, 2020.

[11] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. Directed greybox fuzzing. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, page 2329–2344, New York, NY, USA, 2017. Association for Computing Machinery.

[12] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. *CoRR*, abs/2005.14165, 2020.

[13] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark

Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners, 2020.

[14] Cagatay Catal and Deepti Mishra. Test case prioritization: a systematic mapping study. *Software Quality Journal*, 21(3):445–478, 2013.

[15] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating large language models trained on code, 2021.

[16] Tsong Yueh Chen, Fei-Ching Kuo, Robert G Merkel, and TH Tse. Adaptive random testing: The art of test case diversity. *Journal of Systems and Software*, 83(1):60–66, 2010.

[17] Zimin Chen and Martin Monperrus. A literature study of embeddings on source code, 2019.

[18] Zimin Chen and Martin Monperrus. The remarkable role of similarity in redundancy-based program repair, 2019.

[19] Hyunjin Choi, Judong Kim, Seongho Joe, and Youngjune Gwon. Evaluation of bert and albert sentence embedding performance on downstream nlp tasks, 2021.

[20] Danilo Dessì, Rim Helaoui, Vivek Kumar, Diego Reforgiato Recupero, and Daniele Riboni. TF-IDF vs word embeddings for morbidity identification in clinical notes: An initial study. *CoRR*, abs/2105.09632, 2021.

[21] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding, 2018.

[22] Daniel Di Nardo, Nadia Alshahwan, Lionel Briand, and Yvan Labiche. Coverage-based test case prioritisation: An industrial case study. In *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*, pages 302–311, 2013.

[23] S. Elbaum, A.G. Malishevsky, and G. Rothermel. Test case prioritization: a family of empirical studies. *IEEE Transactions on Software Engineering*, 28(2):159–182, 2002.

[24] Sebastian Elbaum, Gregg Rothermel, and John Penix. Techniques for improving regression testing in continuous integration development environments. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 235–245, 2014.

[25] Robert Feldt and Simon Poulding. Finding test data with specific properties via metaheuristic search. In *2013 IEEE 24th International Symposium on Software Reliability Engineering (ISSRE)*, pages 350–359. IEEE, 2013.

[26] Robert Feldt, Simon Poulding, David Clark, and Shin Yoo. Test set diameter: Quantifying the diversity of sets of test cases. In *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, pages 223–233. IEEE, 2016.

[27] Robert Feldt, Simon Poulding, David Clark, and Shin Yoo. Test set diameter: Quantifying the diversity of sets of test cases. In *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, pages 223–233. IEEE, 2016.

[28] Robert Feldt, Richard Torkar, Tony Gorschek, and Wasif Afzal. Searching for cognitively diverse tests: Towards universal test diversity metrics. In *2008 IEEE International Conference on Software Testing Verification and Validation Workshop*, pages 178–186. IEEE, 2008.

[29] Robert Feldt, Richard Torkar, Tony Gorschek, and Wasif Afzal. Searching for cognitively diverse tests: Towards universal test diversity metrics. In *2008 IEEE International Conference on Software Testing Verification and Validation Workshop*, pages 178–186, 2008.

[30] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. Codebert: A pre-trained model for programming and natural languages, 2020.

[31] Gordon Fraser and Andrea Arcuri. Evosuite: automatic test suite generation for object-oriented software. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pages 416–419, 2011.

[32] Patrice Godefroid, Michael Y. Levin, and David Molnar. Sage: Whitebox fuzzing for security testing: Sage has had a remarkable impact at microsoft. *Queue*, 10(1):20–27, jan 2012.

[33] Patrice Godefroid, Hila Peleg, and Rishabh Singh. Learn&fuzz: Machine learning for input fuzzing, 2017.

[34] Alex Graves and Jürgen Schmidhuber. Framewise phoneme classification with bidirectional lstm and other neural network architectures. *NEURAL NETWORKS*, pages 5–6, 2005.

[35] Jiazhen Gu, Xuchuan Luo, Yangfan Zhou, and Xin Wang. Muffin: Testing deep learning libraries via neural architecture fuzzing, 2022.

[36] Xiaodong Gu, Hongyu Zhang, and Sunghun Kim. Deep code search. ICSE '18, page 933–944, New York, NY, USA, 2018. Association for Computing Machinery.

[37] Xiaodong Gu, Hongyu Zhang, Dongmei Zhang, and Sunghun Kim. Deep api learning. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 631–642, 2016.

[38] Xiaodong Gu, Hongyu Zhang, Dongmei Zhang, and Sunghun Kim. Deep api learning, 2017.

[39] Jacob A. Harer, Louis Y. Kim, Rebecca L. Russell, Onur Ozdemir, Leonard R. Kosta, Akshay Rangamani, Lei H. Hamilton, Gabriel I. Centeno, Jonathan R. Key, Paul M. Ellingwood, Erik Antelman, Alan Mackay, Marc W. McConley, Jeffrey M. Opper, Peter Chin, and Tomo Lazovich. Automated software vulnerability detection with machine learning, 2018.

[40] M.P.E. Heimdahl, D. George, and R. Weber. Specification test coverage adequacy criteria = specification test generation inadequacy criteria. In *Eighth IEEE International Symposium on High Assurance Systems Engineering, 2004. Proceedings.*, pages 178–186, 2004.

[41] Hadi Hemmati. How effective are code coverage criteria? In *2015 IEEE International Conference on Software Quality, Reliability and Security, QRS 2015, Vancouver, BC, Canada, August 3-5, 2015*, pages 151–156. IEEE, 2015.

[42] Hadi Hemmati. Chapter four - advances in techniques for test prioritization. volume 112 of *Advances in Computers*, pages 185–221. Elsevier, 2019.

[43] Hadi Hemmati. Chapter four - advances in techniques for test prioritization. *Adv. Comput.*, 112:185–221, 2019.

[44] Hadi Hemmati, Andrea Arcuri, and Lionel Briand. Achieving scalable model-based testing through test case diversity. *ACM Transactions Software Eng. Methodology*, 22(1), March 2013.

[45] Hadi Hemmati, Lionel Briand, Andrea Arcuri, and Shaukat Ali. An enhanced test case selection approach for model-based testing: An industrial case study. In *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE '10, page 267–276, New York, NY, USA, 2010. Association for Computing Machinery.

[46] Christopher Henard, Mike Papadakis, Mark Harman, Yue Jia, and Yves Le Traon. Comparing white-box and black-box test prioritization. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, pages 523–534. IEEE, 2016.

[47] Jordan Henkel, Shuvendu K. Lahiri, Ben Liblit, and Thomas Reps. Code vectors: Understanding programs through embedded abstracted symbolic traces. ESEC/FSE 2018, page 163–174. Association for Computing Machinery, 2018.

[48] Kexin Huang, Jaan Altosaar, and Rajesh Ranganath. Clinicalbert: Modeling clinical notes and predicting hospital readmission. *CoRR*, abs/1904.05342, 2019.

[49] Rubing Huang, Quanjun Zhang, Dave Towey, Weifeng Sun, and Jinfu Chen. Regression test case prioritization by code combinations coverage, 2020.

[50] Laura Inozemtseva and Reid Holmes. Coverage is not strongly correlated with test suite effectiveness. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, page 435–445. Association for Computing Machinery, 2014.

[51] Rene Just. The major mutation framework: Efficient and scalable mutation analysis for java. 07 2014.

[52] René Just, Darioush Jalali, and Michael D. Ernst. Defects4J: A database of existing faults to enable controlled testing studies for Java programs. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, pages 437–440, San Jose, CA, USA, July 23–25 2014.

[53] Tien-Duy B Le and David Lo. Deep specification mining. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 106–117, 2018.

[54] Guodong Li, Indradeep Ghosh, and Sreeranga P. Rajan. Klover: A symbolic execution and automatic test generation tool for c++ programs. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, *Computer Aided Verification*, pages 609–615, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[55] Zheng Li, Mark Harman, and Robert M Hierons. Search algorithms for regression test case prioritization. *IEEE Transactions on software engineering*, 33(4):225–237, 2007.

[56] Yang Liu and Mirella Lapata. Text summarization with pretrained encoders. *CoRR*, abs/1908.08345, 2019.

[57] David Lo, Siau-Cheng Khoo, Jiawei Han, and Chao Liu. *Mining software specifications: methodologies and applications*. CRC Press, 2011.

[58] Laurens van der Maaten and Geoffrey Hinton. Visualizing data using t-sne. *Journal of machine learning research*, 9(Nov):2579–2605, 2008.

[59] Ehsan Mashhadi and Hadi Hemmati. Applying codebert for automated program repair of java simple bugs. *CoRR*, abs/2103.11626, 2021.

[60] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space, 2013.

[61] Tomas Mikolov, Wen-tau Yih, and Geoffrey Zweig. Linguistic regularities in continuous space word representations. In *Proceedings of the 2013 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 746–751, Atlanta, Georgia, June 2013. Association for Computational Linguistics.

[62] Breno Miranda, Emilio Cruciani, Roberto Verdecchia, and Antonia Bertolino. Fast approaches to scalable similarity-based test case prioritization. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, pages 222–232, 2018.

[63] Debajyoti Mondal, Hadi Hemmati, and Stephane Durocher. Exploring test suite diversification and code coverage in multi-objective test case selection. In *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*, pages 1–10. IEEE, 2015.

[64] Quoc Thai Nguyen, Thoai Linh Nguyen, Ngoc Hoang Luong, and Quoc Hung Ngo. Fine-tuning bert for sentiment analysis of vietnamese reviews, 2020.

[65] T. D. Nguyen, A. T. Nguyen, and T. N. Nguyen. Mapping api elements for code migration with vector representations. In *2016 IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C)*, pages 756–758, 2016.

[66] Tanzeem Bin Noor and Hadi Hemmati. A similarity-based approach for test case prioritization using historical failure data. In *2015 IEEE 26th International Symposium on Software Reliability Engineering (ISSRE)*, pages 58–68. IEEE, 2015.

[67] Tanzeem Bin Noor and Hadi Hemmati. Test case analytics: Mining test case traces to improve risk-driven testing. In *2015 IEEE 1st International Workshop on Software Analytics (SWAN)*, pages 13–16, 2015.

[68] Tanzeem Bin Noor and Hadi Hemmati. Studying test case failure prediction for test case prioritization. PROMISE, page 2–11, New York, NY, USA, 2017. Association for Computing Machinery.

[69] Tanzeem Bin Noor and Hadi Hemmati. Studying test case failure prediction for test case prioritization. In *Proceedings of the 13th International Conference on Predictive Models and Data Analytics in Software Engineering*, PROMISE, page 2–11, New York, NY, USA, 2017. Association for Computing Machinery.

[70] Rongqi Pan, Mojtaba Bagherzadeh, Taher Ahmed Ghaleb, and Lionel Briand. Test case selection and prioritization using machine learning: A systematic literature review. *arXiv preprint arXiv:2106.13891*, 2021.

[71] Matthew E. Peters, Mark Neumann, Mohit Iyyer, Matt Gardner, Christopher Clark, Kenton Lee, and Luke Zettlemoyer. Deep contextualized word representations, 2018.

[72] Jiezhong Qiu, Hao Ma, Omer Levy, Scott Wen-tau Yih, Sinong Wang, and Jie Tang. Blockwise self-attention for long document understanding, 2019.

[73] R Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2020.

[74] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. *CoRR*, abs/1910.10683, 2019.

[75] Murali Krishna Ramanathan, Mehmet Koyuturk, Ananth Grama, and Suresh Jagannathan. Phalanx: a graph-theoretic framework for test case prioritization. In *Proceedings of the 2008 ACM symposium on Applied computing*, pages 667–673, 2008.

[76] Veselin Raychev, Martin Vechev, and Eran Yahav. Code completion with statistical language models. PLDI '14, page 419–428, New York, NY, USA, 2014. Association for Computing Machinery.

[77] Joaquim Santos, Bernardo Consoli, and Renata Vieira. Word embedding evaluation in downstream tasks and semantic analogies. In *Proceedings of the 12th Language Resources and Evaluation Conference*, pages 4828–4834, Marseille, France, May 2020. European Language Resources Association.

[78] Yulei Sui, Xiao Cheng, Guanqin Zhang, and Haoyu Wang. Flow2vec: Value-flow-based precise code embedding. *Proc. ACM Program. Lang.*, 4(OOPSLA), November 2020.

[79] Stephen W Thomas, Hadi Hemmati, Ahmed E Hassan, and Dorothea Blostein. Static test case prioritization using topic models. *Empirical Software Engineering*, 19(1):182–212, 2014.

[80] Foivos Tsimpourlas, Ajitha Rajan, and Miltiadis Allamanis. Supervised learning over test executions as a test oracle. In *Proceedings of the 36th Annual ACM Symposium on Applied Computing*, pages 1521–1531, 2021.

[81] Foivos Tsimpourlas, Gwenyth Rooijackers, Ajitha Rajan, and Miltiadis Allamanis. Embedding and classifying test execution traces using neural networks. *IET Software*, 08 2021.

[82] Michele Tufano, Dawn Drain, Alexey Svyatkovskiy, Shao Kun Deng, and Neel Sundaresan. Unit test case generation with transformers and focal context, 2020.

[83] Eric Wallace, Yizhong Wang, Sujian Li, Sameer Singh, and Matt Gardner. Do NLP models know numbers? probing numeracy in embeddings. *CoRR*, abs/1909.07940, 2019.

[84] Ke Wang, Rishabh Singh, and Zhendong Su. Dynamic neural program embedding for program repair, 2018.

[85] Ke Wang and Zhendong Su. Learning blended, precise semantic program embeddings, 2019.

[86] Ke Wang, Zhendong Su, and Rishabh Singh. Dynamic neural program embeddings for program repair. In *International Conference on Learning Representations*, 2018.

[87] Zhiguo Wang, Patrick Ng, Xiaofei Ma, Ramesh Nallapati, and Bing Xiang. Multi-passage BERT: A globally normalized BERT model for open-domain question answering. *CoRR*, abs/1908.08167, 2019.

[88] Soheila Zangeneh. Test2vec: An execution trace embedding for behaviour coverage analysis of test cases, 2021.

[89] Hong Zhu, Patrick A. V. Hall, and John H. R. May. Software unit test coverage and adequacy. *ACM Computing Surveys*, 29(4):366–427, December 1997.

[90] Xiaogang Zhu, Sheng Wen, Seyit Camtepe, and Yang Xiang. Fuzzing: A survey for roadmap. *ACM Comput. Surv.*, jan 2022. Just Accepted.