Investigating Execution Trace Embedding for Test Case Prioritization

Emad Jabbar[1], Hadi Hemmati[2,*], and Robert Feldt[3]
[1]University of Calgary, Calgary, AB, Canada
[2]York University, Toronto, ON, Canada
[3]Chalmers University of Technology, Gothenburg, Sweden
emad.jabbar@ucalgary.ca, hemmati@yorku.ca, robert.feldt@chalmers.se
*corresponding author

2023 IEEE 23rd International Conference on Software Quality, Reliability, and Security (QRS) | 979-8-3503-1958-3/23/$31.00 ©2023 IEEE | DOI: 10.1109/QRS60937.2023.00036

Abstract—Most automated software testing tasks, such as test case generation, selection, and prioritization, can benefit from an abstract representation of test cases. Although test case representation usually is not explicitly discussed in software literature, but traditionally test cases are mostly represented based on what they cover in source code (e.g., which statements or branches), which is in fact an abstract representation. In this paper, we hypothesize that execution traces of test cases, as representations of their behaviour, can be leveraged to better encode test cases compared to code-based coverage information, for automated testing tasks. To validate this hypothesis, we propose an embedding approach, Test2Vec, based on an state-of-the-art neural program embedding (CodeBert), where the encoder maps test execution traces, i.e. sequences of method calls with their inputs and return values, to fixed-length, numerical vectors. We evaluate this representation in automated test case prioritization (TP) task. Our TP method is a classifier trained on the passing and failing vectors of historical test cases, in regression testing. We compare our embedding with multiple baselines and related work including CodeBert itself. The empirical study is based on 250 real faults and 703,353 seeded faults (mutants) over 250 revisions of 10 open-source Java projects from Defects4J, with a total of over 1,407,206 execution traces. Results show that our approach improves all alternatives, significantly, with respect to studied metrics. We also show that both inputs and outputs of a method are important elements of the execution-based embedding.

Keywords–Test Case Prioritization; Transformers; Software Testing; Embedding;

1. Introduction

Code coverage is one of the most common test adequacy metrics [1], which is used to optimize tasks such as automated test generation [2] and prioritization [3]. However, the common code-based coverage metrics (e.g., statement and branch coverage) are weak criteria in evaluating a test case/suite in terms of detecting bugs. A covered statement in the source code is not necessarily bug free. Depending on which path the execution has taken to land on that statement, the behaviour of the system might be right (passing test case) or wrong (failing test case). This is referred to as path coverage. However, all-path coverage is usually not feasible, given that number of concrete execution paths is often too large (or even infinite).

Recent studies propose to represent a test case with its entire execution path, abstracted at some level (to make it feasible), but still including the order of execution (e.g., as a sequence of method calls, the test covers at run-time) [4], [5]. To make the sequence length manageable, these approaches either cut-off the trace at a fix length or map the traces into a fixed-size vector space, for instance using a one-hot encoding approach [6] over all existing methods in that code base.

In this paper, we propose a novel methodology to represent a test case as an abstract execution path. We then show how we can leverage such representation to prioritize test cases which outperforms existing test prioritization methods based on common code coverage metrics as well as state of the art alternatives to represent execution traces.

The motivation of our work, based on the related literature, is that input and output (I/O) variables are important in testing [4], [5], [7]. Thus the test case should not be represented without them. However, including I/O values increases the dimensionality of the representation space. Thus a simple encoding approach (e.g., one-hot-encoding) is not applicable, and more advanced models are needed to encode/embed test execution traces.

Therefore, in this study, we propose a neural sequence learning model called "Test2Vec", which can create a representation of test case execution. This can potentially be used for multiple downstream testing tasks. However, the focus of this paper is only on one representative tasks of unit test case prioritization (TP).

Our proposed 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 [8] 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 250 faulty releases from 10 open-source Java projects from the Defect4J [9] benchmark dataset, with a total of 758,396 execution traces. We explore two main research questions (RQs) in the experiments. In RQ1, we examine the effectiveness of our embedding architecture compared to five alternatives, both from coverage-based category as well as most advanced embedding architectures. In RQ2, we go deeper

979-8-3503-1958-3/23/$31.00 ©2023 IEEE
DOI 10.1109/QRS60937.2023.00036

into our design choices for the model architecture and show whether including methods' I/O are helpful in increasing the effectiveness or not.

Overall, RQ1 results show that our TP technique leads to significant improvements, in terms of the normalized rank of the first failing test (FFR) and average percentage of fault detection (APFD). The average (and median) FFR relative improvement over the best baseline (CodeBert) is: 40.71% (34.5%). Same improvements with respect to APFD are 3.88% (3.57%). RQ2 results show the impact of I/O in the embedding, which consistently is in favor of including the I/O values. The average (and median) FFR relative decline, if we drop return values from the embedding are 27% (23%), and 87% (110%) if we drop both input and return values.

The contributions of this paper can be summarized as:

- Proposing an embedding approach (Test2Vec) based on CodeBert, but specialized for test case execution trace representation that considers the sequence of method calls alongside with their inputs and outputs.
- Proposing a test case prioritization method, leveraging Test2Vec embedding.
- Conducting a large-scale experiment, comparing the proposed approach with five baselines.
- An ablation study to investigate the effect of methods' I/O in an embedding for the TP task.

All the source code and datasets of this study are available in a public repository [10], for replication.

2. *Test2Vec:* AN EXECUTION TRACE EMBEDDING

The key idea of this paper is a new embedding model to represent execution traces of unit test cases, in the method-call level. Applying our embedding on each execution trace abstracts the test case's run-time behavior and maps it into a vector in an N-dimensional latent space. These embedded vectors can then be used to analyze a test suite as well as an individual test case. 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). One can also use the relative distance between these vectors in a test suite as a mean to identify anomalies.

Thus, we propose "Test2Vec Embedding", 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 down-stream task (TP), we argue that using the pass/fail label as the final objective for fine-tuning makes more sense than the other options.

The specific Test2Vec implementation we propose and evaluate here uses CodeBERT layers. CodeBERT [11] is a BERT variant for both Natural Language (NL) and Programming Language (PL). It learns general-purpose contextual embeddings for both NL and PL. CodeBERT layers are originally pre-trained on generic tasks, such as masked label prediction, on PL and NL data. CodeBERT's job in Test2Vec is to embed method names and I/O parameters, individually, which is the closest sub-task within trace embedding that can leverage the learnt representation from CodeBert's pre-trained model on static source code. 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 on variety of testing-related tasks with potentially a different fine-tuning.

A critical point is that the sequences of called methods 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 contributes the most and thus should be included, per task. However, the method calls, their inputs, and theirs outputs seems like the minimum that are suggested in various literature [12], [13], [3].

We call each triplet of <Outputs, Method Name , 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 1 and Figure 2 show an overview of our approach, which consists of three main phases, as follows:

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. 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 this problem, we use mutation testing to automatically seed artificial faults and generate more failing test cases.

To log the execution traces of the test cases, we have instrumented the SUT code, using AspectJ [14] that allows us to capture the method calls (including nested calls in the source

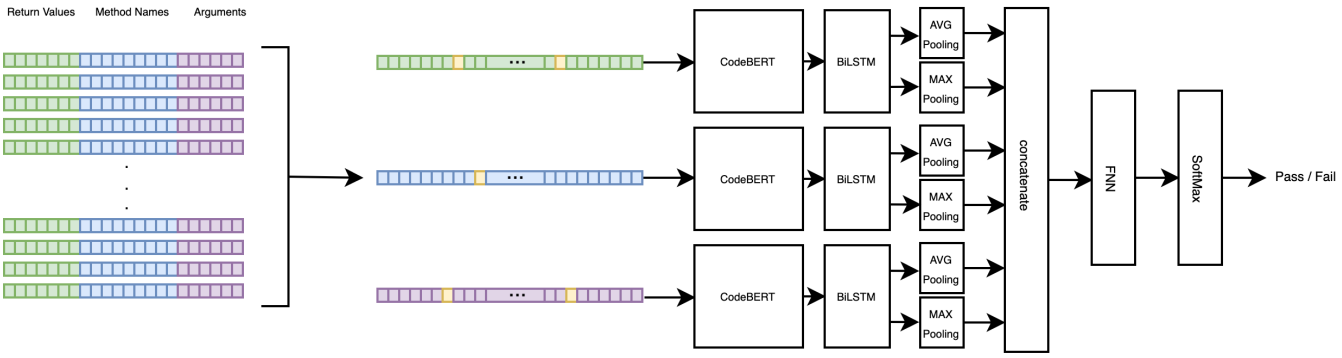

Figure 1. Test2Vec Overall Model Architecture.

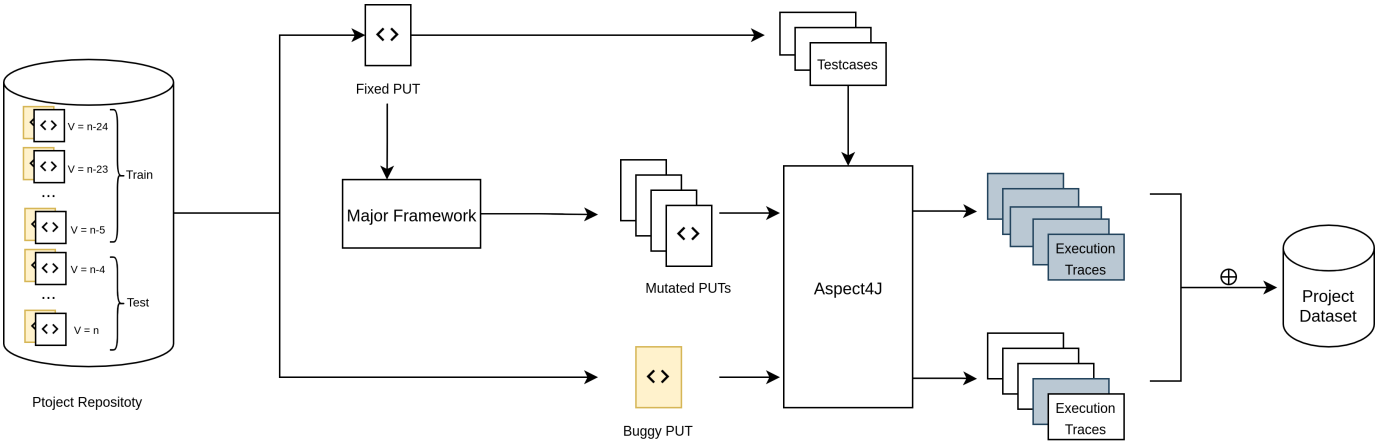

Figure 2. Overall process of generating the trace datasets, per project. "failed" traces (shown as colored boxes) are collected from running tests on mutants or real bugs.

code when the SUT is called) and all I/O names, types, and values at run-time. These implementation details can be found in the publically available repository of the paper.

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 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 (that is usually done with test stubs and mock objects) rather than integration testing of multiple classes.

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 [15] have shown that most embedding techniques cannot work well with numerical data, especially large values. They also demonstrated that language 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 simply break the range into five categories: "negative big value (NBV)", "negative value (NV)", "zero value (ZV)", "positive value (PV)", and "positive big value (PBV)". For $string$ and $array$, we just abstract into empty vs non-empty string or array. More details about these tokens can be found in the replication package [10]. For 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 potentially can improve our results even more. This line of research is in our future work.

Figure 3 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.

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 in a trace (number of calls) to MAX Context. Besides, the

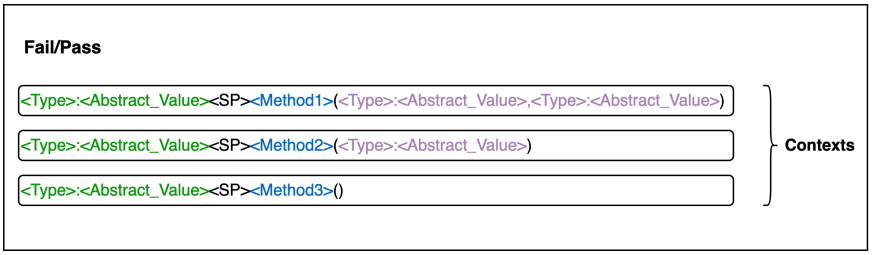

Figure 3. Trace template after pre-processing. Green: Output, Blue: Method Name, Purple: Input(s).

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.

Phase2: Trace embedding generation: After extracting all method calls from each trace (preserving their order), we create three different sequences for outputs, method names, and input arguments (we use $< NO\_ARG >$ in cases where there is no Input or Output). We pass each sequence to a distinct CodeBERT model to generate a separate embedding, per token, in each sequence. 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 per project), while classifying the traces as pass or fail.

We use a small LSTM layer after each Transformer, which is a very common way to extract sentence embeddings from BERT models [16]. At this point, we have a separate embedding for each token in each sequence. We then use Pooling layers [17] 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.

Finally, we concatenate the three embeddings that we calculate for the sequences and use this as the trace embedding.

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 fail or pass for each embedding (learn the 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 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.

3. Evaluation

3.1 Research Questions

RQ1) How effective is Test2Vec-based embedding for test prioritization?

The goal of this RQ is to show that Test2Vec allows a fine-grained representation of test cases which can improve performance on the TP downstream tasks. Thus we compare Test2Vec with alternative representation methods for TP. Our approach (Test2Vec classifier-TP) for test prioritization based on Test2Vec is using classifier (Softmax) layers' probabilities that we train 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 prioritization (state of the practice)? The goal of this sub-RQ is to justify the whole idea of test case behaviour representation compared to simple code coverage, for TP. As a basic coverage-based TP, we use the Greedy-based additional coverage TP [18], 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.

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-based TP with TPs using (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 closest related work.

RQ2) Does the Test2Vec embedding benefit from input/output parameter values in the execution traces, to better prioritize test cases? One key motivation to use embeddings is the ability to reduce high dimensional space (when the trace includes I/O) to a fixed-size space. In RQ1, our Test2Vec embeds all the method calls and their I/O 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 output or both the input and the output parameters from the embedding, correspondingly.

3.2 Evaluation Metrics

The evaluation metrics we have used are standard metrics from the TP domain, such as APFD [19] and the rank of the first failing test [7]. Our FFR metric represents the normalized position of the first failed test case for a test case prioritization method. Although the abbreviation FFR is defined by us, the rank of first failing test is a common metric for TP, specially when there is only one bug to catch. It basically is the number of tests which must be executed before the (first) fault is detected, given the test execution order suggested by the prioritization method. The only modification we made here is normalizing the rank over the size of test suite, given that the test suite sizes are quite different in various projects .

Our second metric, the average percentage of fault detection (APFD), is very common TP metric and 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 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.

APFD is more informative if there are multiple choices of failing test cases and faults to prioritize so that the area under curve is a good "summary" of the prioritization effectiveness. Otherwise, if there is only one bug, simply measuring the rank of 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 FFR 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 case in this case, we use APFD to compare techniques.

Another design choice for the experiments 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). Applying TPs on 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 in the class-level, one must know which class to focus on.

To deal with this challenge we evaluate our approach in 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 the seeded faults). To evaluate the TPs at the class-level, we use the FFR. This way, we get a sense of TPs' effectiveness at both levels. However, our main metric is FFR on the class-level, for the following reasons: (a) It is the most direct metric for ranking, in particular when there is only one fault to catch, (b) We can use it on the real faults in our dataset, and (c) In practice, there are ways to decide which source classes should be tested (e.g., using defect prediction, or simply based on last changes to the code) and thus a TP technique can know in advance which test suite(s) to target.

Therefore, in RQ1, to compare different TP methods, we report AFPD results on all test cases in all test suites, per version, to detect mutants. We also report individual FFRs per faulty class when the goal is to prioritize tests to catch the real bugs in Defect4J. In RQ2, however, due to limited space, we can only report and discuss one metric. So we decided to report FFR here since the evaluation with respect to real faults were more important for us.

3.3 Dataset and the Data Extraction Procedure

In this study, we run our experiments on 10 open-source Java projects from the Defects4J project [9]: Commons-Lang (Lang), Joda-Time (Time), Commons-Cli (Cli), JFreeChart (Chart), Commons-Compress (Compress), Jsoup, Mockito, jackson-databind (JacksonDatabind), jackson-core (Jackson-Core), and commons-math (Math). Defects4j provides multiple revisions per 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 I shows the number of versions and classes per project.

In Defects4J, each faulty version has exactly one bug 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 faults. 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 2 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 mutation tool, the Major [20] 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 provided mutation operators in Major. 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 on, 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.

We augment our dataset with the failing traces on seeded faults, until we have an exact balanced dataset. Given that the total number of failing tests on seeded faults are 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 one 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.

At the end, we had a total of 192,768 execution traces over the 10 projects, ranging from around 2,186 to 82,384 traces, per project. Table I 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 historical 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. This way there is no data leak from past, when evaluating the models on the Test set.

TABLE I
PROJECTS UNDER STUDY.

Project	#Total Traces	#Trace Per Failed Class Min	Median	Max
Chart	82384	57	76	1192
Cli	2994	18	23	170
Jsoup	6798	6	23	132
Compress	8352	7	11.5	17
Lang	15732	11	19	73
Time	38038	54	104	153
JacksonCore	5590	7	8	16
JacksonDatabind	21478	15	15	18
Mokito	2186	7	8	9
Math	9216	7	11	16
Total	192,768			

3.4 Design

To answer **RQ1-1**, we implemented two TP methods: (a) an Additional Greedy Algorithm [3] 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 $P(failing \mid trace)$ 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: 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.

An important note to mention is that we deliberately did not choose an advanced recent coverage-based TP such as a search-based optimization based approach since the hypothesis to verify in this paper is about the encoding phase of the TP (static coverage vs dynamic embedding) and not the optimization part. To be a fair comparison, Test2Vec TP is also using the simplest classifier. In other words, improving the optimization phase of one of the baselines (e.g. an optimized coverage-based or a better classifier or a few-shot learner for Test2Vec) would create confounding factors in the experiment (i.e., we would not know if the differences are due to the tests representation or the optimization).

The goal of **RQ1-2** is to compare Test2Vec with the current body of research [11], [12] 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 [12]. 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 1). 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 average pooling strategies and concatenate these vectors together. 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's, 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**) are 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 [4].

Regarding the FFR analysis, all 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 8 out of 50 failures (one version from Cli, one from Compress, two from JacksonCore, one from JacksonDatabind, and three from Mokito).

Regarding the APFD analysis, the traces are collected from the mutants as explained in 3.2) so there is no concern about the limited number of data point, as in the FFR analysis.

Given that all TPs have a random component (even in coverage-based-TP techniques that are deterministic there are many test cases with the same coverage score, where we had to break the tie randomly), in both RQs, we execute each TP 30 times per version and record the median values (AFPD and FFR) of those 30 runs for that version. To report results per project, we show the mean and median of these medians for five versions of each project. To compare two TPs in any RQ, we look at the mean, median, and statistical significant test results of FFRs and APFDs, over the total 50 (5×10) project versions.

We use a paired (paired over projects and versions) non-parametric statistical significance test (Wilcoxon signed rank test [21] with p-value less than 0.05) and calculated the effect size (rank-biserial correlation [21]), every time we compare two distributions of ranks over their 50 samples.

3.5 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 have 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 fine-tuned for 10 epochs. Training and evaluation of the deep learning model was 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 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 classifier-TP techniques.

3.6 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?

TABLE II
FFRs FOR RQ1, OVER ALL VERSIONS UNDER STUDY, GROUPED BY PROJECTS. EACH FFRs PER VERSION IS THE MEDIAN OF 30 RUNS. THE BOLD VALUES ARE THE DOMINATING RESULTS, PER PROJECT AND TOTAL.

	Test2Vec		Line Coverage		Branch Coverage		One-Hot		LSTM-based		CodeBERT	
	Median	AVG	Median	AVG	Median	AVG	Median	AVG	Median	AVG	Median	AVG
Chart	**5.70**	**7.73**	25.00	31.67	56.67	43.39	18.29	17.37	26.89	25.22	12.28	15.12
Cli	**12.87**	**13.70**	36.37	41.89	44.65	43.14	29.79	30.04	34.65	30.58	17.42	18.05
Jsoup	**13.04**	**16.27**	18.18	21.89	46.81	44.94	27.27	33.68	24.13	35.27	30.43	28.50
Compress	**14.28**	**14.67**	69.04	54.53	48.82	49.21	15.96	21.82	34.82	38.20	20.53	23.92
Lang	**10.52**	**12.00**	45.61	57.22	50.90	47.34	27.28	35.41	31.57	27.61	18.18	22.39
Time	**9.25**	**10.69**	20.54	21.87	20.64	25.29	24.07	22.51	16.34	19.16	14.42	14.72
Mokito	**18.25**	**18.25**	73.00	73.00	25.78	25.78	26.98	26.98	36.50	36.50	25.39	25.39
Math	**12.5**	**15.89**	48.125	48.66	47.29	47.09	31.25	28.50	28.57	33.87	22.22	22.99
JacksonDatabind	**8.88**	**9.44**	60.00	56.38	56.57	66.16	37.50	41.18	36.66	32.50	24.44	28.88
JacksonCore	**12.50**	**13.09**	28.57	24.10	34.28	30.52	20.28	22.60	31.25	44.94	18.75	19.34
Total	**12.50**	**12.86**	36.93	41.33	47.05	43.27	27.50	27.27	28.57	31.39	19.09	21.69

TABLE III
APFDs FOR RQ1, OVER ALL VERSIONS UNDER STUDY, GROUPED BY PROJECTS. EACH APFD PER VERSION IS THE MEDIAN OF 30 RUNS. THE BOLD VALUES ARE THE DOMINATING RESULTS, PER PROJECT AND TOTAL.

	Test2Vec		Line Coverage		Branch Coverage		One-Hot		LSTM-based		CodeBERT	
	Median	AVG	Median	AVG	Median	AVG	Median	AVG	Median	AVG	Median	AVG
Chart	**92.71**	**94.37**	69.85	71.21	73.02	73.45	79.99	80.16	76.93	78.82	90.15	91.21
Cli	**93.07**	**91.32**	71.79	73.89	69.17	68.22	76.23	78.93	81.19	78.21	88.31	87.74
Jsoup	**90.81**	**90.57**	68.96	70.83	70.54	69.92	75.93	74.17	76.93	77.23	90.17	90.44
Compress	89.76	**89.13**	75.01	74.21	69.67	70.61	77.30	79.36	81.27	81.09	**94.43**	88.53
Lang	**89.54**	**88.74**	72.31	72.18	71.89	72.79	76.41	75.49	72.94	73.62	82.19	86.49
Time	**92.35**	**91.70**	74.82	76.53	73.47	72.28	76.64	78.92	75.07	74.32	85.92	86.30
Mokito	**89.17**	**89.83**	69.58	69.09	69.95	70.53	81.87	80.01	80.11	80.15	82.58	84.72
Math	**92.00**	**91.42**	69.47	70.14	68.74	68.06	79.52	80.52	74.31	77.19	91.01	90.17
JacksonDatabind	**88.68**	**88.64**	71.11	72.19	69.51	68.17	75.65	73.92	75.41	76.71	81.34	81.29
JacksonCore	**92.73**	**92.48**	70.04	68.10	70.46	71.50	80.61	81.94	70.98	71.53	88.19	87.92
Total	**91.40**	**90.82**	70.57	71.83	70.20	70.55	76.97	78.34	76.17	76.88	88.25	87.43

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 II, among all 42 (50 minus the 8 versions that did not have enough tests in the failing test suite) faulty versions under study, the normalized rank of the first failing test (FFR) in the original faulty versions for the Test2Vec classifier-TP has an average of 12.86 and a median of 12.50. In comparison, the line coverage(LC)-based TP has an average of 41.33 and a median of 36.93, and the branch coverage(BC)-based TP technique has an average of 43.27 and a median of 47.05. The relative improvement of the Test2Vec over the coverage-based TPs are: 68.89% (average) and 66.15% (median) over LC-based TP; and 70.28% (average) and 73.43% (median) over BC-based TP.
Tabel III shows the APFD for all execution traces (including failing traces generated by mutation) among all 50 versions under study. the APFD for the Test2Vec classifier-TP has an average of 90.82 and a median of 91.40. In comparison, the line coverage(LC)-based TP has an average of 71.83 and a median of 70.57, and the branch coverage(BC)-based TP technique has an average of 70.55 and a median of 70.20. The relative improvement of the Test2Vec over the coverage-based TPs are: 26.42% (average) and 29.51% (median) over LC-based TP; and 28.72% (average) and 30.19% (median) over BC-based TP.
Running the statistical test on the FFR and APFD confirms that the differences between Test2Vec classifier-TP and the two baselines are statistically significant (with close to 1.0 effect size and negligible p-values).
Table II 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 II also shows the FFR for the one-hot, CodeBERT, and LSTM-based TP methods. The one-hot TP method has an average of 27.27 and a median of 27.50, the CodeBERT TP technique has an average of 21.69 and a median of 19.09, and the LSTM-based method has an average of 31.41 and a median of 28.57. The relative improvement of the Test2Vec over the encoding and embedding baseline TPs are: 54.16% (average)

and 53.24% (median) over One-Hot TP; 34.54% (average) and 40.7% (median) over CodeBERT TP; and 56.25% (average) and 59.06% (median) over LSTM-based TP.

For the APFD experiment, as Table III suggests, the APFD for the one-hot TP method has an average of 78.34 and a median of 76.97, the CodeBERT TP technique has an average of 87.48 and a median of 88.25, and the LSTM-based method has an average of 76.88 and a median of 76.17. This means that the relative improvement of the Test2Vec over the encoding and embedding baseline TPs are: 15.92% (average) and 18.75% (median) over One-Hot TP; 3.81% (average) and 3.57% (median) over CodeBERT TP; and 18.12% (average) and 20.00% (median) over LSTM-based TP.

Running the statistical tests on the FFRs and APFDs again results in above 0.9 effect size and negligible p-values, when comparing our approach with any of the three baselines in this sub-RQ.

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 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.

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

As discussed in section 3.4, 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. We also explained that due to space limits we only report FFRs which are more meaningful to evaluate detecting the real faults in Defect4J (See Section 3.2).

Table IV 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. Looking at these results as an ablation study the relative decline of FFR values of Test2Vec when excluding output values in the embedings are 27%(mean) and 23%(median). The relative decline when we omit both inputs and outputs are dramatic: 87% (mean) and 110%(median).

Similar to the previous RQ, statistical tests on these FFRs show very high effect size and negligible p-values, when comparing our approach with the two variations.

What these results imply is that both inputs and outputs values are significantly important to distinguish pass and fail traces and an embedding that consider them is a better predictive model.

3.7 Discussion on the limitations

The first limitation of our approach is that we need a training set of execution traces for fine tuning. Depending on the testing task and context, this may not be available. But for the test prioritization task in the context of regression testing, this limitation is minor (there are already existing tests). One future direction is to examine more recent large language models such as GPT-4 [22] to embed traces without fine-tuning, or with a little few-shot learning by in-context prompting. This can potentially eliminate this limitation. However, at the time of conducting this study GPT-4 or 3.5 were not available and the older GPTs were not very different than BERT-type models (still needed fine-tuning).

Another limitation is related to cost. One cost issue is to create mutants, which are needed if the real bugs are limited. Another cost is the training cost. However, these tasks (training and mutation generation) are NOT frequent tasks. In many cases it should be enough to do them once per project. As long as there aren't many changes to the project, the trained model and the mutants can be reused. Therefore, the cost of mutation and training can likely be amortized over many versions of a project, reducing the per-version cost. 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.

One limitation of the experiment (not the approach itself) is we implemented the TPs using simple optimization (Greedy for coverage and Softmax for classifier). As explained, we did this deliberately to only focus on the test representation's (embedding's) effect not the optimization, to avoid confounding factors. Future work can further investigate the effect of more advanced optimizations on top of these embeddings and compare them with their advanced counterparts in the coverage-based context.

Finally, we used CodeBERT which has the 512 tokens limitation for each sequence. There are many works on the domain of NLP for handling larger sequences [23], [24], and as explained future work can investigate replacing CodeBERT with a large language model such as GPT-4.

3.8 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 to avoid confounding factors! We wanted metrics that work on both real-world bugs as well as mutations.

TABLE IV
FFRs FOR RQ2, OVER ALL VERSIONS UNDER STUDY, GROUPED BY PROJECTS. EACH FFRs PER VERSION IS THE MEDIAN OF 30 RUNS. THE BOLD VALUES ARE THE DOMINATING RESULTS, PER PROJECT AND TOTAL.

	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

We also wanted to be able to evaluate our TPs both on the whole project version as well as test suite level. The 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. We normalized it since without normalization large and small test sets results where not comparable. The APFD, on the other hand, is used 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.

Regarding conclusion validity, we repeat our training on 10 different projects and report the statistical tests and effect size across multiple versions. We also repeat the TPs for 30 runs, per version. We then carefully reported both median and mean, as well as p-values and effect sizes, to better analyze the differences and their statistical an practical significance.

Finally, in terms of external validity, although we tried to mitigate the threat by selecting multiple (10) 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 specially industrial ones, without further replications.

4. RELATED WORK

In this section, we discuss about some of the background and related works in representation learning in software engineering and in software testing. For the sake of brevity, we don't not discuss all test case prioritization literature, and only include them if they use a novel test representation.

4.1 Sequence Learning in Software Engineering

In software engineering, sequence learning and embedding [25], [26] has been used for tasks such as code completion[27], program repair [28], [29], [30], API learning [31], code search [32], and bug and security vulnerabilities prediction [33]. The focus of learning in these works is on the source code [34] and their crucial drawback, in our context, is missing the execution information, which is one the most important concerns in testing.

In the category of customized embeddings for source code (neural program embedding), Code2Vec [35], Code2Seq [36], Flow2Vec [37], and CodeBERT [11] are among the state-of-the-art. Although these models' underlying architectures defer and new models are being proposed frequently, all the existing pre-trained code models so far are working on the program source code and miss the execution information of traces.

There is however a small but growing community that work on representing dynamic executions. In this context, Henkel *et al.* [38] use 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.* [39] 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 [40], for the task of learning representation vectors for the method's body, the authors show that a mix of symbolic and concrete execution traces outperform other existing approaches that ignore dynamic behavior of the program. These work are among our motivating studies that though applied on different tasks but showed that dynamic traces including input data may be beneficial in testing.

4.2 Test Case Representation for Prioritization

In this section, we review the TP literature from the lens of test representation (i.e., literature with novel optimization but traditional representation are excluded)

Using coverage information: Early works on TP techniques mostly used coverage data of test cases to represent each test case. Elbaum *et al.* [41] 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.* [42] 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 represented as a text string of its code/script. Then the similarity or distance of test cases are measured by applying simple string distance metrics (without applying any encoding or embedding). For instance, Miranda **et al.** [43] used string distance metrics, such as Hamming distance and Levenshtein diversity, to measure similarity/distance between test cases.

Using input data: These works 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.* [44] have leveraged neural statistical models to automatically infer input grammars for Fuzz testing.

Using extra static data: This category contains works that use extra data which are still static, such as: source code, specification models, comments, and ASTs in order to represent test cases. For example, Hemmati *et al.* [45] encode tests based on specification coverage of each test case on the manually drafted UML state machine representation of the system. Mondal *et al.* [6] used one-hot encoding to represent each test case based on a set of method calls that are extracted from each test case, without executing the code. Thomas *et al.*[46] use a topic modeling approach on the source code and comments.

Using dynamic traces of test cases: This is the category that our paper falls in where our execution traces are dynamically collected. Using dynamic data for software testing is not explored enough by researchers. Noor and Hemmati [7] 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.* [47] 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.* [12] 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 paper.

5. Conclusion and Future Work

In this paper, we proposed a novel neural embedding model (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 representation of rich test case information such as the ordering of method calls as well as specific test input and output values. Empirical evaluation, on ten projects from Defects4J, show that our approach outperforms traditional code coverage metrics, state-of-the-art neural program embedding models, and an embedding with other sequence learning approaches, on a test prioritization task.

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 behaviour, on system, security, and performance testing.

References

[1] H. Zhu, P. A. V. Hall, and J. H. R. May, "Software unit test coverage and adequacy," *ACM Computing Surveys*, vol. 29, no. 4, p. 366–427, Dec. 1997.

[2] G. Fraser and A. 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*, 2011, pp. 416–419.

[3] H. Hemmati, "Chapter four - advances in techniques for test prioritization," ser. Advances in Computers, A. M. Memon, Ed. Elsevier, 2019, vol. 112, pp. 185–221.

[4] R. Feldt, R. Torkar, T. Gorschek, and W. Afzal, "Searching for cognitively diverse tests: Towards universal test diversity metrics," in *IEEE International Conference on Software Testing Verification and Validation Workshop*, 2008, pp. 178–186.

[5] T. B. Noor and H. Hemmati, "Test case analytics: Mining test case traces to improve risk-driven testing," in *2015 IEEE 1st International Workshop on Software Analytics (SWAN)*, 2015, pp. 13–16.

[6] D. Mondal, H. Hemmati, and S. 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)*. IEEE, 2015, pp. 1–10.

[7] T. B. Noor and H. Hemmati, "Studying test case failure prediction for test case prioritization," ser. PROMISE. New York, NY, USA: Association for Computing Machinery, 2017, p. 2–11.

[8] A. Graves and J. Schmidhuber, "Framewise phoneme classification with bidirectional lstm and other neural network architectures," *NEURAL NETWORKS*, pp. 5–6, 2005.

[9] R. Just, D. Jalali, and M. 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)*, 2014, pp. 437–440.

[10] Anonymous, "Test2vec repository," Aug. 2023. [Online]. Available: https://anonymous.4open.science/r/Test2Vec/README.md

[11] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, and M. Zhou, "Codebert: A pre-trained model for programming and natural languages," 2020.

[12] F. Tsimpourlas, G. Rooijackers, A. Rajan, and M. Allamanis, "Embedding and classifying test execution traces using neural networks," *IET Software*, 08 2021.

[13] R. Feldt, R. Torkar, T. Gorschek, and W. Afzal, "Searching for cognitively diverse tests: Towards universal test diversity metrics," in *2008 IEEE International Conference on Software Testing Verification and Validation Workshop*, 2008, pp. 178–186.

[14] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold, "An overview of aspectj," in *ECOOP 2001 — Object-Oriented Programming*, J. L. Knudsen, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 327–354.

[15] E. Wallace, Y. Wang, S. Li, S. Singh, and M. Gardner, "Do NLP models know numbers? probing numeracy in embeddings," *CoRR*, vol. abs/1909.07940, 2019.

[16] Q. T. Nguyen, T. L. Nguyen, N. H. Luong, and Q. H. Ngo, "Fine-tuning bert for sentiment analysis of vietnamese reviews," 2020. [Online]. Available: https://arxiv.org/abs/2011.10426

[17] H. Choi, J. Kim, S. Joe, and Y. Gwon, "Evaluation of bert and albert sentence embedding performance on downstream nlp tasks," in *IEEE International conference on pattern recognition (ICPR)*, 2021, pp. 5482–5487.

[18] R. Huang, Q. Zhang, D. Towey, W. Sun, and J. Chen, "Regression test case prioritization by code combinations coverage," 2020. [Online]. Available: https://arxiv.org/abs/2007.00370

[19] R. Pan, M. Bagherzadeh, T. A. Ghaleb, and L. Briand, "Test case selection and prioritization using machine learning: A systematic literature review," *arXiv preprint arXiv:2106.13891*, 2021.

[20] R. Just, "The major mutation framework: Efficient and scalable mutation analysis for java," 07 2014.

[21] R Core Team, *R: A Language and Environment for Statistical Computing*, R Foundation for Statistical Computing, Vienna, Austria, 2020. [Online]. Available: https://www.R-project.org/

[22] OpenAI, "Gpt-4 technical report," 2023.

[23] J. Qiu, H. Ma, O. Levy, S. W.-t. Yih, S. Wang, and J. Tang, "Blockwise self-attention for long document understanding," 2019. [Online]. Available: https://arxiv.org/abs/1911.02972

[24] I. Beltagy, M. E. Peters, and A. Cohan, "Longformer: The long-document transformer," *arXiv preprint arXiv:2004.05150*, 2020.

[25] M. Allamanis, E. T. Barr, P. Devanbu, and C. Sutton, "A survey of machine learning for big code and naturalness," *ACM Computing Surveys (CSUR)*, vol. 51, no. 4, p. 81, 2018.

[26] Z. Chen and M. Monperrus, "A literature study of embeddings on source code," 2019. [Online]. Available: https://arxiv.org/abs/1904.03061

[27] V. Raychev, M. Vechev, and E. Yahav, "Code completion with statistical language models," ser. PLDI '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 419–428.

[28] K. Wang, Z. Su, and R. Singh, "Dynamic neural program embeddings for program repair," in *International Conference on Learning Representations*, 2018.

[29] E. Mashhadi and H. Hemmati, "Applying codebert for automated program repair of java simple bugs," in *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*, 2021, pp. 505–509.

[30] Z. Chen and M. Monperrus, "The remarkable role of similarity in redundancy-based program repair," 2019.

[31] X. Gu, H. Zhang, D. Zhang, and S. Kim, "Deep api learning," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2016, pp. 631–642.

[32] X. Gu, H. Zhang, and S. Kim, "Deep code search," ser. ICSE '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 933–944.

[33] J. A. Harer, L. Y. Kim, R. L. Russell, O. Ozdemir, L. R. Kosta, A. Rangamani, L. H. Hamilton, G. I. Centeno, J. R. Key, P. M. Ellingwood, E. Antelman, A. Mackay, M. W. McConley, J. M. Opper, P. Chin, and T. Lazovich, "Automated software vulnerability detection with machine learning," 2018.

[34] M. Allamanis, E. T. Barr, P. Devanbu, and C. Sutton, "A survey of machine learning for big code and naturalness," *ACM Comput. Surv.*, vol. 51, no. 4, Jul. 2018.

[35] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, "code2vec: Learning distributed representations of code," 2018.

[36] U. Alon, S. Brody, O. Levy, and E. Yahav, "code2seq: Generating sequences from structured representations of code," 2019.

[37] Y. Sui, X. Cheng, G. Zhang, and H. Wang, "Flow2vec: Value-flow-based precise code embedding," *Proc. ACM Program. Lang.*, vol. 4, no. OOPSLA, Nov. 2020.

[38] J. Henkel, S. K. Lahiri, B. Liblit, and T. Reps, "Code vectors: Understanding programs through embedded abstracted symbolic traces," ser. ESEC/FSE 2018. Association for Computing Machinery, 2018, p. 163–174.

[39] K. Wang, R. Singh, and Z. Su, "Dynamic neural program embedding for program repair," 2018.

[40] K. Wang and Z. Su, "Learning blended, precise semantic program embeddings," 2019.

[41] S. Elbaum, A. Malishevsky, and G. Rothermel, "Test case prioritization: a family of empirical studies," *IEEE Transactions on Software Engineering*, vol. 28, no. 2, pp. 159–182, 2002.

[42] D. Di Nardo, N. Alshahwan, L. Briand, and Y. Labiche, "Coverage-based test case prioritisation: An industrial case study," in *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*, 2013, pp. 302–311.

[43] B. Miranda, E. Cruciani, R. Verdecchia, and A. Bertolino, "Fast approaches to scalable similarity-based test case prioritization," in *IEEE/ACM International Conference on Software Engineering (ICSE)*, 2018, pp. 222–232.

[44] P. Godefroid, H. Peleg, and R. Singh, "Learn&fuzz: Machine learning for input fuzzing," *CoRR*, vol. abs/1701.07232, 2017.

[45] H. Hemmati, A. Arcuri, and L. Briand, "Achieving scalable model-based testing through test case diversity," *ACM Transactions Software Eng. Methodology*, vol. 22, no. 1, Mar. 2013.

[46] S. W. Thomas, H. Hemmati, A. E. Hassan, and D. Blostein, "Static test case prioritization using topic models," *Empirical Software Engineering*, vol. 19, no. 1, pp. 182–212, 2014.

[47] M. K. Ramanathan, M. Koyuturk, A. Grama, and S. Jagannathan, "Phalanx: a graph-theoretic framework for test case prioritization," in *Proceedings of the 2008 ACM symposium on Applied computing*, 2008, pp. 667–673.