Test2Vec: Learning Representations of Test Cases

Quan Sun
Electrical and Computer Engineering Department
University of Calgary
Calgary, Canada
quan.sun@ucalgary.ca

***Abstract*—The automation of test case generation is a long-term research topic in test automation. Meanwhile, with the rise of machine learning and deep learning for the past years, embedding on source code has attracted many researchers' attention. How to represent test cases might be an important step in test case generation. This project applies the deep learning model code2vec to test cases which are represented by latent vectors and this process can be called test2vec. Two experiments are conducted: the first one uses test cases' names as labels to train test2vec and conducts prediction and analogy trials; and the second one replaces test cases' names with classes' names, trains test2vec using classes' names as labels, and compares similarities within and between classes. The first experiment shows the success of transferring code2vec to test2vec, and the second experiment indicates that test2vec can separate different classes but has difficulty in separating the failing test case from passing test cases by using the metric cosine similarity and data visualization techniques. In addition, test2vec can separate non-test methods from test cases.**

Keywords—test2vec, test case, code2vec, deep learning

I. INTRODUCTION

In the software development life cycle, software testing is a time-consuming and costly task. Test automation is a promising solution to reduce the cost of testing and increase the quality of the testing process within a limited budget, which can be applied to different testing activities such as test case generation, test prioritization, test execution, and test repair. In software testing, test case generation is a main step. During the past few years, researchers have been working on different tools and methods to improve testing efficiency and optimize testing time [1]. It has been found that many proposed test case generation tools and methods can contribute to reduce testing time and decrease the cost of resources required during testing phase. The existing literature proposes diverse tools and methods that support procedural and object oriented languages, such as Java. These tools and methods can benefit automation of test case generation.

With the rise of machine learning, especially deep learning and artificial neural networks, more and more researchers have been working on the application of machine learning to software testing, such as manual test case failure prediction and test case prioritization. But to the best of my knowledge, very few researchers have applied artificial neural network to test case generation [2]. I will propose a Natural Language Processing approach to represent test cases based on code2vec [3]. Many researches have proposed different approaches about machine learning models including embeddings on source code [4, 5] and as the first step, I will study the code2vec model and check the performance of this model [3]. With the project going, I will train the test2vec model using test classes' names as labels . Here for this course project, I will focus on representing test cases similar to code2vec, which can be called test2vec. This approach will be helpful to our future research towards working for the automation of test case generation. This technique will also benefit academic researchers who are working in the field of software test automation.

The contributions of this project will be as follows:

- Decompose test cases to collections of paths in their abstract syntax trees (ASTs) [3]
- Build the test2vec model based on the code2vec model
- Train the test2vec model and generate test case representations (test vectors)
- Apply similarity functions to test vectors and check the similarity between passing test case vectors and failing test case vectors and different classes as well
- Test vector dimensionality reduction and visualization

II. BACKGROUND

In this section I will cover backgrounds on test case generation, source code embedding and data visualization techniques.

A. Test Case Generation

The goal of testing is to find errors or bugs in a system or program. A set of test cases is called a test suite. Test case generation is the process of generating new test suites for a particular system. Usually people try to generate a test suite which satisfies a given criterion, e.g., code coverage. No matter in the traditional developing manner or in recent rapid releasing environments, due to the limited budget, manually generating test cases and exhaustive testing of the subject system are usually infeasible, but engineers would like to increase the likelihood that the generated test suite uncovers errors or bugs in the subject system. Automatic test case generation can accelerate the testing process and generate high-quality test suites by utilizing specific techniques. Automatic test case generation is the act of using system to identify truth table of tests based on a set of contexts and outcomes. Automatic test case generation helps tester speed up testing cycles but with reduced effort and cost. It also plays a role in maintaining test cases and increases the efficiency of software testing techniques.

B. Abstract Syntax Tree (AST)

Here I describe the representation of a code snippet/test case as a set of syntactic paths in its abstract syntax tree (AST). AST is used to handle whole snippets of code and act as input to the test2vev model. In order to clearly define AST, three definitions are copied from the code2vec paper [3].

Definition1(Abstract Syntax Tree). An Abstract Syntax Tree (AST) for a code snippet C is a tuple $\langle N, T, X, s, \delta, \phi \rangle$ where N is a set of nonterminal nodes, T is a set of terminal nodes, X is a set of values, $s \in N$ is the root node, $\delta : N \rightarrow (N \cup T)^*$ is a function that maps a nonterminal node to a list of its children, and $\phi : T \rightarrow X$ is a function that maps a terminal node to an associated value. Every node except the root appears exactly once in all the lists of children.

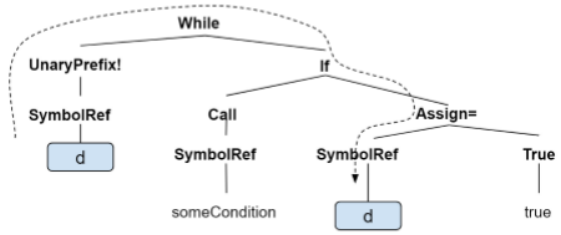
Definition2(AST path). An AST-path of length k is a sequence of the form: $n_1 d_1 ... n_k d_k n_{k+1}$, where $n_1, n_{k+1} \in T$ are terminals, for $i \in [2..k]$: $n_i \in N$ are non-terminals and for $i \in [1..k]$: $d_i \in \{ \uparrow, \downarrow \}$ are movement directions (either up or down in the tree). If $d_i = \uparrow$, then: $n_i \in \delta(n_{i+1})$; if $d_i = \downarrow$, then: $n_{i+1} \in \delta(n_i)$. For an AST-path p, we use start(p) to denote n1 - the starting terminal of p; and end(p) to denote n_{k+1} - its final terminal.

Definition 3 (Path-context). Given an AST Path p, its path-context is a triplet $\langle x_s, p, x_t \rangle$ where $x_s = \phi(\text{start}(p))$ and $x_t = \phi(\text{end}(p))$ are the values associated with the start and end terminals of p.

Fig. 1 is an example of AST [6] which uses a simple JavaScript program. This program is parsed into an AST which shows the structure of this program, and then AST can be decomposed into several path contexts. The dashed line is a path context, that is, [d, SymbolRef ↑ UnaryPrefix! ↑ While ↓ If ↓ Assign= ↓ SymbolRef, d]. Here d and d are called values and the string in the middle part is called path. One path context consists of two values and one path.

```
while (!d) {
  if (someCondition()) {
    d = true;
  }
}
```

(a) A simple JavaScript program.



(b) The program's AST, and example to an AST path.

Fig. 1. AST example

C. Source code embedding

With the success of word embedding techniques, researchers are paying more and more attention to programming languages and building many novel applications of source code. In this proposal, test cases are kind of source code and the similar techniques which have been applied to source code can be applied to test cases. The test2vec model will be very similar to code2vec model [3] and may do some modification in the process of the project, which is shown in Fig. 2.

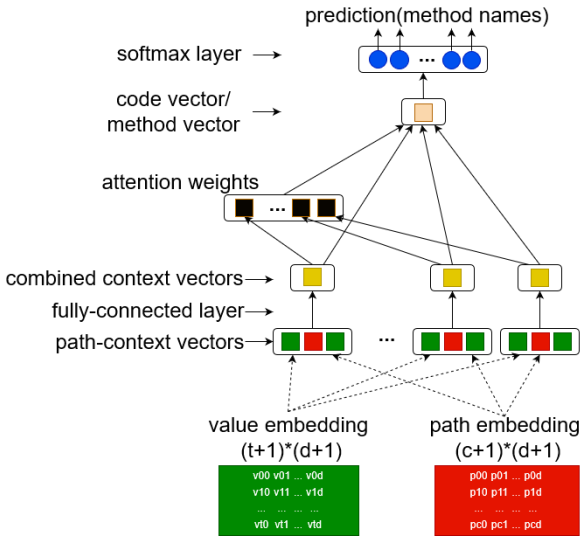


Fig. 2. Overview of the code2vec model

This model is a neural network model based on Natural Language Processing techniques word2vec and attention mechanism. The path contexts generated from one method's AST cannot be fed directly into the model and need to be converted to vectors where embedding comes in. In Fig.1, green color means value embedding and red color means path embedding. Each value (and path as well) in this model is represented by a d+1 length vector. One path context concatenates two values vectors and one path vector to make up one 3*(d+1) length vector. In Fig.1, each square means a d+1 length vector. All these path context vectors are fed into a fully connected layer and the tanh activation function transfer them to combined context vectors. Attention weights are assigned to each combined context vector and then sum them to get code vector/test vector. Attention weights are initialized randomly and updated during the model training process. The generated code vector can be the representation of the input method and then it will be fed into a softmax layer and be transformed to predictions which are probabilities of method names. The higher one probability of one method is, the more similar this method with the input method is. The model can be modified to use fail or pass as labels based on whether test cases fail source codes or not. The goal of this is to train the model to capture the difference between failing test cases and passing test cases.

D. t-SNE

t-SNE (t-distributed Stochastic Neighbor Embedding) is a tool to visualize high-dimensional data in a low-dimensional space. It converts similarities between data points (here test vectors) to joint probabilities and tries to minimize the divergence between the joint probabilities of the low-dimensional embedding and the high-dimensional data. t-SNE has a cost function that is not convex, i.e. with different initializations we can get different results, which is a disadvantage of t-SNE.

E. PCA

PCA(principal component analysis) is a statistical procedure that uses an orthogonal transformation to convert a set of observations of possibly correlated variables (entities each of

which takes on various numerical values) into a set of values of linearly uncorrelated variables called principal components. PCA is used to explain the variance-covariance structure of a set of variables through linear combinations. It is often used as a dimensionality-reduction technique.

III. METHODOLOGY

In this section, I will explain the approach of test2vec in detail.

A. Test Case Decomposition

Decompose a test case to a collection of paths in their abstract syntax trees (ASTs). This part involves both methods and their input values. For test cases, one important component that triggers failure is the input values. Based on AST, a collection of path-contexts will be generated. The path-context consists of two parts: the assertion method paths and the input values, which is similar to definition 3. This work will derive value vocabulary matrix and path vocabulary matrix which will be used in next part.

B. Test2vec Model

Build the test2vec model based on the code2vec model. Hyperparameters might need to be tuned based on test case dataset. The labels will be test cases' names and classes' names. Then train the test2vec model and generate test case representations (test vectors).

C. Calculate Similarity and Data Visulization

Apply cosine similarity function to test vectors and check the similarity between passing test case vectors and failing test case vectors. Apply t-SNE/PCA to reduce dimensionality of test vectors and visualize test cases, which will help check the similarities between passing test case vectors with failing test cases.

$$sim(pi,\ pj) = \frac{pi \cdot pj}{\|pi\|\|pj\|} \quad (1)$$

$$Sim(pi,\ P) = \frac{1}{n}\sum_{j=1}^{n} sim(pi,\ pj) \quad (2)$$

$$SIM(T) = \frac{1}{t}\sum_{i=1}^{t} Sim(pi,\ T - pi) \quad (3)$$

The three equations above are used in the experiment part. Equation 1 is the standard cosine similarity function for two points/vectors pi and pj, which is represented by two vectors' dot product divided by the product of two vectors' magnitudes. Equation 2 is the similarity between one point pi and a set of points P, which is represented by the mean value of similarities between pi and each point in P. Equation 3 is the similarity within a set of points T, which is represented by the mean value of similarities from any one point to the rest of points in T.

IV. EXPERIMENT

A. Objective and Research Questions

The objective of this research is to check whether there is difference between test cases and whether this difference can be captured by test2vec.

RQ1: can test2vec capture any difference between passing test cases and failing test cases in the context of test vectors?

The reason to propose this research question is that the values in the failing test cases are very special which might separate them from passing test cases.

RQ2: can test2vec capture any difference between different classes?

The experiment for this research question is to train the model using classes' names as labels. The training process might generate obvious differences between different classes.

RQ3: does data visualization benefit the detection of the difference if there is difference?

For our model, each test case is represented by a high dimensional vector whose dimension needs to be reduced in order to be visualized. t-SNE and PCA will be considered for this.

B. Software and Dataset

In this project, I first replicate code2vec using test cases as the input. I use six different Java projects of the defects4j database [7]. The database provides 438 faults and around 40,000 Junit tests from six different open-source Java projects. All the faults are real, reproducible and have been isolated in different versions. There is a faulty version and a fixed version of the program source code, for each fault. The faulty source code is modified in the fixed version to remove the fault. The test cases are the same in both of the faulty and the fixed versions. There is at least one test case in each version that fails on the faulty version but passes on the fixed version.

TABLE I. DEFECTS4J

Identifier	Project name	Number of bugs
Chart	JFreeChart	26
Closure	Closure compiler	176
Lang	Apache commons-lang	65
Math	Apache commons-math	106
Mockito	Mockito	38
Time	Joda-Time	27

For the three research questions, I focus on one buggy version lang_1b of the project Lang , which consists of 91 test classes and around 2000 developer-written test cases. I use Randoop to generate more than 32000 random test cases for this buggy version (around 350 test cases per class). The final used dataset consists of 34055 test cases (around 274 test cases per class). I divide the total test cases into three datasets: training (27031, 79.4%), validation (6015, 17.7%) and test (1009, 3%). I don't leave too many test cases in the test dataset because I only keep developer-written test cases in it and just use it to check the generation of the trained model. And average 10 test cases per class are enough to answer the three research questions. For training and validation stage, the input of test2vec is the generated test cases by Randoop plus part of developer-written test cases; for test stage, the input of test2vc is the remaining part of developer-written test cases. Because I use lang_1b as an

example, I have only one failing developer-written test case which should be used in test stage. (I don't know whether the generated test cases fail or pass, which doesn't matter). I checkout lang 1b and find that the only one failing test case is TestLang747 in Class NumberUtilsTest. Therefore, the Class NumberUtilsTest is very special, and I divide part of Class NumberUtilsTest test cases (including Randoop-generated test cases) into training and validation dataset, and test dataset. The test case TestLang747 is in test dataset.

Note: in this project, the terms 'point' and 'test case' are the same meaning and they can be used alternatively.

C. Evaluation Measurement

Because test cases are represented by vectors, cosine similarity function will be used to calculate the similarity between passing test cases and failing test cases. Another measurement approach is to track the similarity to see how it changes between failing and passing tests for the faulty class and whether adding the failing test vector to passing one will change the average similarity between vectors. The detailed equations are listed in Methodology C part.

D. Procedure

I run these two experiments through Google Colab, and the programming language and packages are listed below:

TABLE II. PLATFORM & DEPENDENCIES

Programming language/ package	Version
Python	3.6.8
TensorFlow-GPU	1.15.0
Keras	2.2.5
CUDA	10.0
cdDNN	7.6.3

I use GPU to speed up model training.

For the first experiment, its goal is to replicate the original work of code2vec and to check whether test2vec can run empirically well. This experiment procedure is as follows:

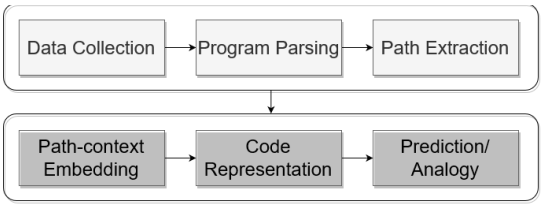


Fig. 3 Overview of the steps for the first experiment

The whole procedure can be divided into two big parts: data extraction and modelling. These two parts can be done separately in practice.

Step 1 Data Collection: collect all test cases from all 6 projects of Defects4j.

Step 2 Program Parsing: parse each test case into its corresponding AST.

Step 3 Path Extraction: extract path contexts for each test case based on its AST.

```
File: my_dataset.test.raw.txt
Average total contexts: 490.1029207232267
Average final (after sampling) contexts: 165.26008344923505
Total examples: 719
Empty examples: 0
Max number of contexts per word: 5007
File: my_dataset.val.raw.txt
Average total contexts: 537.4744186046512
Average final (after sampling) contexts: 151.12403100775194
Total examples: 645
Empty examples: 0
Max number of contexts per word: 15692
File: my_dataset.train.raw.txt
Average total contexts: 280.7605424484488
Average final (after sampling) contexts: 110.7066426050264
Total examples: 37681
Empty examples: 0
Max number of contexts per word: 45035
Dictionaries saved to: data/my_dataset/my_dataset.dict.c2v
```

Fig. 4 Information of Path Extraction

This figure above shows three datasets to extract path-contexts: test, validation and training datasets. 39045 (719+645+37681) test cases are used, and the maximum number of path contexts per test case is 45035 which is very huge. After sampling, generally less than 200 path contexts are used for each test case in this model.

Step 4 Path-context Embedding: based on all path contexts of all test cases build the value vocabulary and path vocabulary and at the beginning initialize them randomly.

Step 5 Code Representation: path-context vectors are fed into and train the test2vec model with test cases' names as labels to enable the ability of representing input methods as code vectors.

```
Model: "model"
__________________________________________________________________________________________________
Layer (type)                    Output Shape         Param #     Connected to
==================================================================================================
input_1 (InputLayer)            [(None, 200)]        0
__________________________________________________________________________________________________
input_2 (InputLayer)            [(None, 200)]        0
__________________________________________________________________________________________________
input_3 (InputLayer)            [(None, 200)]        0
__________________________________________________________________________________________________
token_embedding (Embedding)     (None, 200, 128)     7894656     input_1[0][0]
                                                                 input_3[0][0]
__________________________________________________________________________________________________
path_embedding (Embedding)      (None, 200, 128)     54105472    input_2[0][0]
__________________________________________________________________________________________________
concatenate (Concatenate)       (None, 200, 384)     0           token_embedding[0][0]
                                                                 path_embedding[0][0]
                                                                 token_embedding[1][0]
__________________________________________________________________________________________________
dropout (Dropout)               (None, 200, 384)     0           concatenate[0][0]
__________________________________________________________________________________________________
time_distributed (TimeDistribut (None, 200, 384)     147456      dropout[0][0]
__________________________________________________________________________________________________
input_4 (InputLayer)            [(None, 200)]        0
__________________________________________________________________________________________________
attention (AttentionLayer)      ((None, 384), (None, 384         time_distributed[0][0]
                                                                 input_4[0][0]
__________________________________________________________________________________________________
target_index (Dense)            (None, 21507)        8258688     attention[0][0]
==================================================================================================
Total params: 70,406,656
Trainable params: 70,406,656
Non-trainable params: 0
```

Fig. 5 Model summary

Figure 5 shows the summary of modelling: this model uses the embedding technique, dropout regularization and attention mechanism. This step trains 70,406,656 parameters all together.

Step 6 Prediction/Analogy: the test dataset can be fed to get the code vectors and the prediction; code vectors can be used to conduct the task of test cases analogy recovery. This step mimics the analogy in the paper [3].

Steps 1-3 belong to the data extraction part and steps 4-6 belong to the modelling part.

For the second experiment, it uses classes' names as labels and checks the similarities within and between classes. The experiment procedure is as follows:

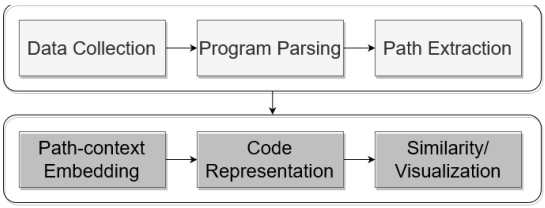


Fig. 6 Overview of the steps for the second experiment

The whole procedure is very similar to the first experiment, but there are three main differences: step 1, step5 and step 6.

For step 1 data collection, because there are only around 2000 test cases (developer-written ones) for lang_1b, in order to train the model, more test cases are needed. I use Randoop to generate more than 32000 test cases randomly and automatically. In addition, classes' names are used as labels, so I replace test cases' names with their corresponding classes' names. Randoop command for one test class Mutable is as follows:

java -classpath E:\randoop-4.2.1\tbin;E:\randoop-4.2.1\randoop-all-4.2.1.jar randoop.main.Main gentests --testclass=org.apache.commons.lang3.mutable.Mutable --regression-test-basename=Mutable --junit-package-name=Mutable --maxsize=10 --time-limit=60 --junit-output-dir=E:\randoop-4.2.1\testsuite2 --randomseed=2

For step 5 code representation, I do hyperparameter tuning. Because there are only 91 classes' names which are labels, the embedding size 128 is large and I tune the embedding size for value and path from 128 to 32. In addition, I train the model for 30 epochs and finally F1 scores become very stable, shown in Fig.7. I select the model of epoch 29 with the highest validation score as the best model, shown in Fig.8. The metric used is F1 score.

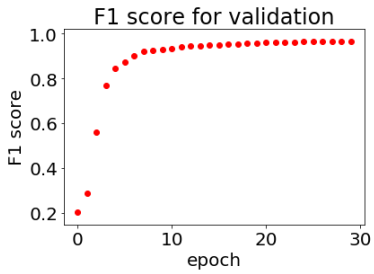


Fig. 7. F1 score for validation

```
Saved after 29 epochs in: models/test2vec/saved_model_iter29
Loading test data from: data/my_dataset/my_dataset.val.c2v
Done loading test data
Done testing, epoch reached
Evaluation time: 0H:0M:3S
After 29 epochs: Precision: 0.9672990329962353, recall: 0.9658018867924528, F1: 0.9665498801401439
```

Fig. 8. Scores for epoch 29

One thing to say is that methods' names are all based on Camel Case in the dataset and during the phase of path extraction, each method name is parsed into lowercase sub tokens separated with vertical bar lines. For example, the method stringUtils is parsed into string|utils. Based on this, the calculation of F1 score is not trivial. If the true label is string|escape|utils, and the prediction is string|utils, then the model doesn't say the prediction is totally wrong. Instead, the model counts the sub tokens. Here, true positive = 2, false positive = 0, true negative = 0, false negative = 1. Then precision = tp/(tp+fp) = 2/(2+0) = 1 and recall = tp/(tp+fn) = 2/(2+1) = 2/3. Finally, F1 = 2*precision*recall/(precision+recall) = 0.8 which is much higher than 0 (0 means totally wrong). Intuitively, this way to calculate F1 score is much more accurate than taking the whole method name as a unit.

For step 6, this experiment conducts similarity calculation within and between classes. And then use t-SNE and PCA to visualize similarities.

To answer RQ1, the failing test class including the trigger test case are extracted and similarities from each test case to the rest test cases are calculated. The similarity from the trigger test case to the rest test cases can be compared with similarities from any other test case to the rest. In addition, the similarity of the set of all test cases except the trigger one can be compared with the similarity of all test cases including the trigger one to check whether the trigger one decreases the similarity.

To answer RQ2, the similarity within each test class and similarities between the different two test classed will be calculated and compared.

To answer RQ3, after representing each test case with its corresponding code vector, t-SNE and PCA can be used to conduct dimensionality reduction and visualize the low-dimensional data.

E. Results and Discussion

1) The first experiment

Trained model is used to predict the test dataset. An example is shown below.

```
Original name:  test|interpolation
        (0.161903) predicted: ['test', 'interpolation']
        (0.159949) predicted: ['compute', 'value']
        (0.097878) predicted: ['test', 'get', 'typed', 'percent']
        (0.092176) predicted: ['test', 'cluster']
        (0.089894) predicted: ['test', 'sphere']
        (0.085329) predicted: ['test', 'contains', 'char']
        (0.083055) predicted: ['get', 'model', 'function', 'jacobian']
        (0.080698) predicted: ['compute', 'jacobian']
        (0.077936) predicted: ['test', 'plane']
        (0.071182) predicted: ['handle', 'step']
        Attention:
        0.336810        context1
        0.112377        context2
        0.070368        context3
        0.048505        context4
        0.044920        context5
        0.044825        context6
        0.028144        context7
        0.027473        context8
        0.020858        context9
        0.017606        context10
```

Fig. 9. Prediction example

The first part of this figure shows the prediction of method names and the value before each predicted method name is the probability. Top 1 method name is the most similar one and in this case the prediction is correct. The second part of this figure indicates attention weights to each context. Here, I just use context+number to represent each path context because the real path contexts are very long and redundant for the purpose of illustration.

After prediction, the analogy is explored using Python gensim package. The examples are as follows:

Table III. ANALOGY EXAMPLES

Input method	Similar method names & cosine similarity
'test|int'	('test|long|array', 0.878) ('test|double', 0.860) ('test|float|array', 0.859)
'test|constructor'	('test|constructor|rp|ri', 0.849) ('test|copy', 0.837) ('test|blank', 0.837)
'test|int'+'test|long' -'test|float'	('test|long|array', 0.785) ('test|append|super', 0.769) ('test|operate', 0.754)

Note: in programming, each method has been represented by a vector; in practice, top 10 methods names will be listed and here only 3 are listed for illustration.

For 'test|int', the top1 is 'test|long|array', which makes a little bit sense. For 'test|constructor', the top1 is 'test|constructor|rp|ri', which makes more sense. For 'test|int'+'test|long'-'test|float', the top1 is 'test|long|array' and one interesting thing is that: for 'test|int', the third one is 'test|float|array', but for 'test|int'+'test|long'-'test|float', 'test|float|array' no longer shows in top 10 methods. From this interesting analogy trials, the model works fine so far.

2) The second experiment

a) About RQ1

For the class NumberUtils whose test case TestLang747 is the trigger test case, I used its 35 test cases in the test dataset, and I calculate the average cosine similarity from one point(test case) to the rest 34 points(test cases) in this class and sorted them below (descending). The bold value below is the average cosine similarity from TestLang747 to the rest points. We guess that test2vec might separate it from the other points (test cases) and the similarity value should be at the end of this list. But in this case below, this assumption is not correct.

[0.9207586122264065, 0.9175380650586896, 0.9174295246523304, 0.9163765106239047, 0.9162559574143009, 0.9156122235480398, 0.9141436800272033, 0.9139347090264138, 0.9137434533074253, 0.91239768206972, 0.9123300061513785, **0.9114366831487047**, 0.9110098049649971, 0.9105794695342938, 0.9097740899906818, 0.9086601160669643, 0.9080405799768085, 0.9078976886941365, 0.9033223832374602, 0.8985325833581848, 0.8980979479570339, 0.8978625932791882, 0.8978575276472209, 0.8978455252444884, 0.8977616378378462, 0.8977608845951333, 0.8977414037998082, 0.8976773658979228, 0.8976728540248792, 0.8913836966801103, 0.8658260577610588, 0.7633684741702454, 0.5044977549492135, 0.45691144161559144, 0.33887512646969536]

In addition, I also calculate similarities within the test class NumberUtils without and with TestLang747 using Equation (3) and the results are 0.858 and 0.861, which means that adding the failing test case improves the similarity.

I doublecheck the TestLang747 test case and find there are a lot of other assertions which don't trigger failures. So, the result above might derive from that the other assertions can generate many other path-contexts which dilute the influence of the trigger assertion which is enclosed by a red rectangle below.

Fig. 10. TestLang747

b) About RQ2

In this part, I use the failing test class NumberUtils and select another three passing test class randomly to illustrate the results. So, here I will compare four test classes: NumberUtils, StrSubstitutor, SerializationUtils and ToStringBuilder, whose information is shown in TABLE IV.

TABLE IV. INFORMATION OF USED TEST CLASSES

Test Class	# points/test cases
NumberUtils	35
StrSubstitutor	19
SerializationUtils	12
ToStringBuilder	45

Note: the test cases of each test class are the ones in the test dataset and not all test cases from the original test class, because test cases of developer-written test classes have been split into training, validation and test dataset.

Firstly, Equation 1 is used to calculate similarities between two points within each class and between two classes. 10 boxplots are generated to show distributions of similarities of four test classes and all pairs as shown in Fig. 11. The first four boxplots are similarity distributions of each individual test class and the other 6 boxplots are similarity distributions of each pair of two test classes. For example, the boxplot of NumberUtils is generated by collecting all similarities between any two test cases in NumberUtils, and the boxplot of NumberUtils&StrSubstitutor is generated by collecting all similarities between any two test cases in the collection of all test cases from NumberUtils and StrSubstitutor.

For the first four boxplots, we can see that the similarities of these four test classes are ranked in the descending order as follows: SerializationUtils, StrSubstitutor, NumberUtils and ToStringBuilder. Points in ToStringBuilder are more disperse than the other three test classes.

By comparing the rest 6 boxplots with the first four individual test classes' boxplots, we can clearly find that the

median similarity value of any combination of two test classes is smaller than that of each individual test class. Let's take NumberUtils&StrSubstitutor as an example. Its median similarity value is around 0.2, but median similarity values of both NumberUtils and StrSubstitutor are greater than 0.9. NumberUtils&StrSubstitutor includes not only similarity values which are already in NumberUtils and StrSubstitutor but also similarity values of two test cases from NumberUtils and StrSubstitutor respectively. Therefore, the boxplot of NumberUtils&StrSubstitutor is extended compared with NumberUtils or StrSubstitutor, but still the highest similarity value is the same with that of both individual test classes.

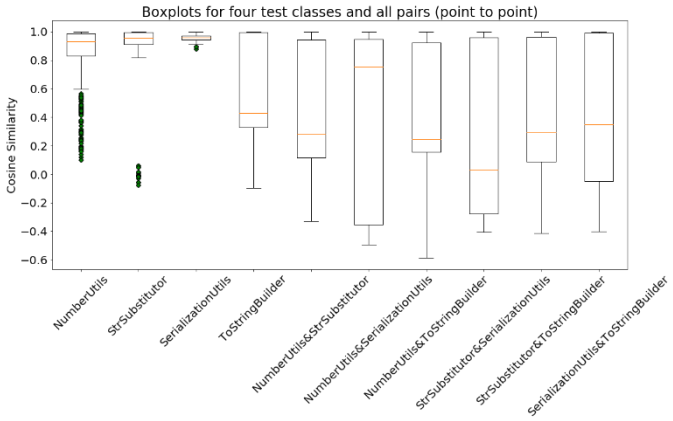


Fig. 11. Similarities from point to point

In order to show the influence of pairing two test classes more clearly, Equation 2 is used to calculate similarities between any one point and the rest points. I use the same four test classes as shown in Fig. 12.

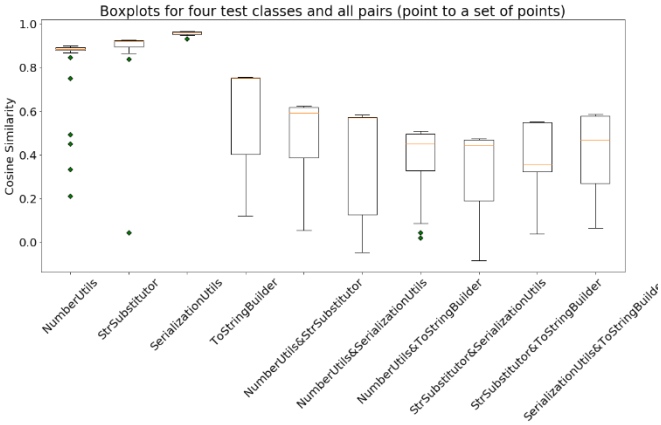


Fig. 12. Similarities from point to a set of points

For the first four boxplots, we can draw the same result with the last figure that the similarities of these four test classes are ranked in the descending order as follows: SerializationUtils, StrSubstitutor, NumberUtils and ToStringBuilder. Points in ToStringBuilder are more disperse than the other three test classes.

Comparing the last 6 boxplots with the first four, most pairs' similarities are lower than each individual test class' similarities. In terms of median similarity values, all pairs' similarities are lower than each individual test class' similarity, which indicates that different classes are separated from each other in the context of cosine similarity. For each point in pairing test classes, its similarity value not only considers the rest points in the same class but also considers the points in the other class, which will make this point's similarity value smaller.

Lastly, let's use Equation 3 to calculate similarities within each individual test class and pairing test classes as shown in Fig. 13.

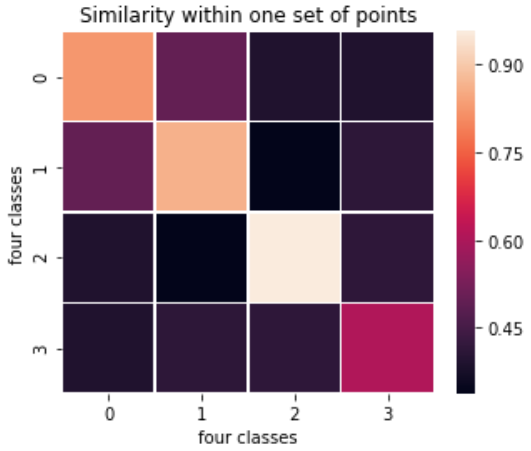


Fig. 13. Heatmap for similarities of four classes

The tick labels '0','1','2' and '3' are corresponding to NumberUtils, StrSubstitutor, SerializationUtils and ToStringBuilder. The lighter the color is, the more similar the point set is. The figure above shows that the diagonal similarity values are larger than the off-diagonal ones and this result is consistent with those generated by two previous figures.

c) About RQ3

Each point is represented by a 96-dimensional vector. t-SNE and PCA are used to reduce each point's dimension and then plot them in 2D space which are shown in Fig. 14 and 15.

At the beginning, I consider only using t-SNE to reduce the dimensionality of code vectors. During the process, I find that t-SNE is not trustworthy and different hyperparameters can generate very different 2D plots. Therefore, I also use PCA which is less flexible to reduce the code vectors' dimensionality. Using both dimensionality reduction techniques can confirm the results generated from plots.

From the two figures below, we can see that both figures are very consistent:

- The trigger test case is in the cluster of NumberUtils and cannot be separated from the other points in the same class.
- The four classes are separated from each other obviously and form four clusters.
- Similarities of these four classes are ranked from large to small: SerializationUtils, StrSubstitutor, NumberUtils and ToStringBuilder.
- The points of ToStringBuilder are more diverse than those of the other three classes.

We can see that these two plots derive the same results with those of RQ1 and RQ3. Therefore, we can say that data visualization helps.

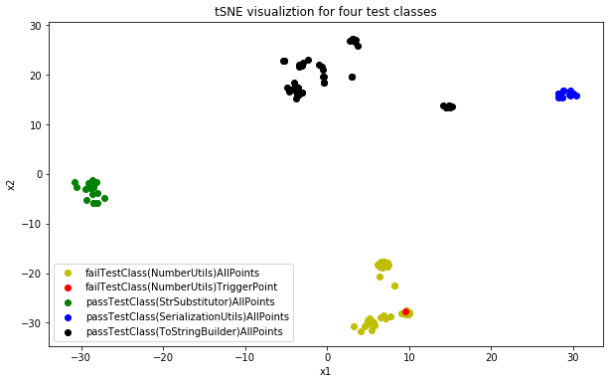


Fig. 14. t-SNE visualization for four test classes

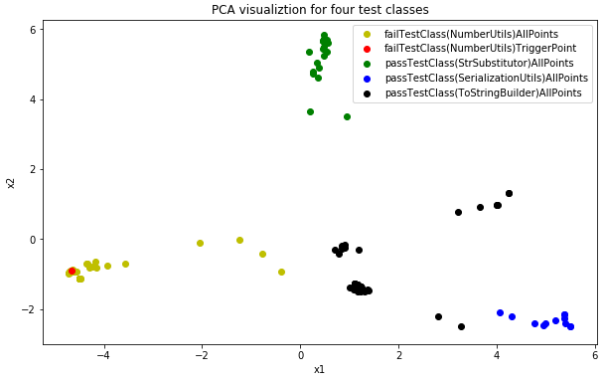


Fig. 15. PCA visualization for four test classes

d) One interesting observation

In boxplots of 'About RQ2' part, we can see green diamond points in several test classes, and I use green diamond to represent outliers. I manually check these outliers and find that the outliers with the smallest similarities are non-test methods (e.g. private/protected methods).

The outlier method for NumberUtils:

```
private boolean checkCreateNumber(final String val) {
    try {
        final Object obj = NumberUtils.createNumber(val);
        if (obj == null) {
            return false;
        }
        return true;
    } catch (final NumberFormatException e) {
        return false;
    }
}
```

The outlier method for StrSubstitutor:

```
@Override
protected String resolveVariable(final String variableName,
    assertEquals("name", variableName);
    assertSame(builder, buf);
    assertEquals(3, startPos);
    assertEquals(10, endPos);
    return "jakarta";
}
```

In order to train the model using classes' names as labels, I make a tricky modification about each test java file: replace each test case' name based on the annotation '@Test' with its class' name. Therefore, I miss some non-test methods and leave them in the java files. In the prediction phase, code vectors of these methods will be generated by the model. Because their method names are non-test or their path contexts are clearly different than those of test cases, the model can generate their code vectors which are separable from test cases' code vectors. So, this interesting observation proves that the model can separate non-test methods from test cases.

e) One note

Beside using test cases' names and classes' names as labels, I also try using fail and pass as labels to train the model according to my project proposal. For this experiment, there are two main drawbacks because of which I haven't done much work here. Firstly, the dataset is hugely unbalanced. Take lang_1b as an example, there are around 2000 test cases among which there is only one failing test case. The model will always predict 'pass'. Secondly, if oversampling technique is used to solve the imbalanced problem, the model might overfit the failing test case and won't generalize very well.

F. Threats to the validity of the study

For conclusion validity, the original paper of code2vec uses 12M methods to train the model, while for my project I only use no more than 40K test cases for each experiment. Due to the time and resource limitation, for the second experiment, I only use one project version as an example. In addition, in response to RQ2 and RQ3, only four test classes are used. Therefore, the used data might not be large enough, and the results are not robust or statistically significant.

For construct validity, I use cosine similarity to measure the similarity between points. There might be other alternatives or dimensionality reduction/feature selection might be done first. This topic needs more study.

For external validity, I have conducted the empirical study based on six real-world open source java libraries from defects4j database with several versions and faults (for the first experiment, I use all six java libraries, and for the second experiment, I only take lang_1b as an example). Most test cases used in the second experiment are generated by Randoop, not developer written. Therefore, generalizing the results to different types of systems/projects may still require further experiments.

V. RELATED WORK

In this section, I discuss the related work in two areas and at the end of each area I discuss how this study is different than the related work.

A. Test Case Generation

The automation of test case generation is a promising approach which helps tester speed up testing cycles but with reduced effort and cost. It also plays a role in maintaining test cases and increases the efficiency of software testing techniques.

Shunhui et.al [2] presented a standard neural network based test case generation for data-flow oriented testing. Some

researchers works on genetic algorithm based test case generation testing [8-11]. Yao et.al [12] proposed a neural network based approach for test data generation of path coverage oriented testing. Mahboubeh Dadkhah [13] applied semantic technology to facilitate the test case generation process. P Mani et.al [14] presented an automatic technique with several programmable logic controller devices with generated test cases achieving high test coverage. Feng Yang [15] studied on manned spacecraft test case generation based on integration method. Zhenzhen et.al [16] proposed a software test case automatic generation technology based on the modified particle swam optimization algorithm. Annibale[17] summarizes challenges and possible solutions in white-box search-based techniques for test case generation. João [18] proposed a novel knowledge discovery metamodel-based unit test cases generation.

Different than all these predecessors' works, our work will propose a NLP-based test case representation approach which will integrate the spectrum of test case generation.

B. Embedding on Source Code

Word embedding techniques such as word2vec have gained great success and based on this, researchers are paying more and more attention to programming languages and building many novel applications of source code.

Based on the literature study of embeddings on source code [4], source code embeddings can be categorized into five groups: embedding of tokens, embedding of functions or methods, embedding of sequences or sets of method calls, embedding of binary code and other embeddings.

Based on word2vec, Chen et al. [19] generate java token embedding to find the correct ingredient in automated program repair. Cosine similarity on the embeddings is utilized to compute a distance between pieces of code, which is the same with my evaluation metric. Similarly, Harer et al. [20] also use word2vec to generate word embedding for C/C++ tokens for software vulnerability prediction. They use the token embedding to initialize a TextCNN model to conduct classification tasks. Allamanis et al. [21] use a context model to generate an embedding for method names. They define a local context which captures tokens around the current token, and a global context which is composed of features from the method code. Their technique is able to generate new method names never seen in the training data. Alon et al. [3] compute Java method embeddings for predicting method names, which inspires my project. Abstract syntax tree is used to construct a path representation between two leaf nodes. Bag of path representations are all aggregated into one single embedding for a method. Pradel & Sen [22] propose to use code snippet embeddings to identify potential buggy code. To generate the training data, they collect code examples from a code corpus and apply simple code transformation to insert an artificial bug. Embedding for positive and negative code examples are generated and they are used to train a classifier. Nguyen et al. [23] explore embeddings for Java and C# APIs and used it to find similar API usage between the languages. They use word2vec for generating API element embeddings, and based on known API mappings, they can find cluster with similar usage across two languages. Redmond et al.[24] explore binary instruction embedding across architectures. Binary instruction embeddings are learned by cluster similar instructions in the same architecture, and preserve the semantic relationship between binary instruction in different architectures. Zuo et al. [25] compute binary instruction embedding to calculate the similarity between two binary blocks. Binary block embeddings are trained on x86-64 and ARM instructions, and instructions in a basic block are combined using LSTM. A Siamese architecture is adopted to determine the similarity between two basic blocks. Allamanis et al. [26] use a graph neural network to generate embedding for each node in an abstract syntax tree. The graph is constructed from nodes and edges from the AST, and then enriched with additional edges indicating data flow and type hierarchies.

Though the related work above is very inspiring for my project, none of them is specific to test cases. Test cases are also code snippets/source code, but they have their own specific properties. Based on AST definitions of the code2vec model, new AST structures specific to test cases will be proposed. The code2vec model will also convert to the test2vec model which means that the model will do some modification.

In this project, I apply code2vec which belongs to the category of embedding of functions or methods to the new datasets (test suites), which is called test2vec. I use test2vec to generate test vector which can be used to separate different test classes.

VI. CONCLUSION

To the best of my knowledge, representing test cases using NLP techniques hasn't been done by researchers so far. Learning representations of test cases properly might benefit testing automation, such as the automation of test case generation.

In this project, I use test2vec by borrowing the code2vec technique from embedding on source code techniques and conduct two experiments. The first experiment is replicating code2vec using test cases from all projects of defects4j as the data and test cases' names as labels. Results of prediction and analogy show that test2vec can be done well. The second experiment is applying test2vec by using classes' names as labels and the main results are as follows:

- Test2vec has difficulty in separating the failing test case from the passing test cases;
- Test2vec can separate different classes;
- Test2vec can separate non-test methods from test cases;
- Data visulization helps.

In the future, more developer-written data/test cases need to be collected and better computation resources will be used to train the test2vec model better. Moreover, I will apply test2vec to other projects from defects4j and other systems/libraries using classes' names as labels which might show more insights and get more robust response to the three research questions.

REFERENCES

[1] R. Singh, A. Singhrova, R. Bhatia. Test Case Generation Tools-A Review[J]. International Journal of Electronics Engineering, 2018, 10(2):586-596.

[2] S. Ji, Q. Chen, P. Zhang. Neural Network Based Test Case Generation for Data-Flow Oriented Testing[C]. 2019 IEEE International Conference On Artificial Intelligence Testing (AITest), 2019:35-36.

[3] U. Alon, M. Zilberstein, O. Levy, and E. Yahav. code2vec: Learning distributed representations of code. Proceedings of the ACM on Programming Languages, 3(POPL):40, 2019.

[4] Z. Chen, M. Monperrus. A Literature Study of Embeddings on Source Code. arXiv:1904.03061 [cs, stat], 2019.

[5] M. Allamanis, E. T Barr, P. Devanbu, C. Sutton. A survey of machine learning for big code and naturalness. arXiv preprint arXiv:1709.06182, 2017.

[6] U. Alon, M. Zilberstein, O. Levy, and E. Yahav. A General Path-based Representation for Predicting Program Properties. In Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 404-419, 2018.

[7] https://github.com/rjust/defects4j, [Online; last accessed 11-Dec-2019].

[8] A.S. Ghiduk, M.J. Harrold. Using Genetic Algorithm to Aid TestData Generation for Data-Flow Coverage[C]. In Proc. of 14th Asia-Pacific Software Engeering Conference, Aichi, Japan, 2007.

[9] Y.X. Chen. Research of Data Flow Based Automatic Test Data Generation Technology[D]. Nanjing University of Posts and Telecommunications, 2011(in Chinese).

[10] S. Varshney, M. Mehrotra. Automated Software Test Data Generation for Data Flow Dependencies using Genetic Algorithm[J]. International Journal of Advanced Research in Computer Science and Software Engineering, 2014, 4(2):472-479.

[11] R. Khan, M. Amjad. Introduction to Data Flow Testing with Genetic Algorithm[J]. International Journal of Computer Applications, 2017, 170(5):39-45.

[12] XJ Yao, DW Gong, B Li. Evolutional Test Data Generation for Path Coverage By Integrating Neural Network[J]. Ruan Jian Xue Bao/Journal of Software, 2016,27(4):828-838(in Chinese).R. Nicole, “Title of paper with only first word capitalized,” J. Name Stand. Abbrev., in press.

[13] M. Dadkhah. Semantic-Based Test Case Generation[C]. 2016 IEEE International Conference on Software Testing, Verification and Validation (ICST), 2016:377-378.

[14] P Mani, M Prasanna. Automatic test case generation for programmable logic controller using function block diagram[C]. 2016 International Conference on Information Communication and Embedded Systems (ICICES), 2016.

[15] F. Yang. Study on Manned Spacecraft Test Case Generation Based on Integration Method[C]. 2018 2nd IEEE Advanced Information Management,Communicates,Electronic and Automation Control Conference (IMCEC), 2018:123-127.

[16] Z. Wang ; Q. Liu. A Software Test Case Automatic Generation Technology Based on the Modified Particle Swarm Optimization Algorithm[C]. 2018 International Conference on Virtual Reality and Intelligent Systems (ICVRIS), 2018:156-159.

[17] A. Panichella. Beyond Unit-Testing in Search-Based Test Case Generation: Challenges and Opportunities [C]. 2019 IEEE/ACM 12th International Workshop on Search-Based Software Testing (SBST), 2019:7-8.

[18] J. Pires, F. Abreu. Knowledge Discovery Metamodel-Based Unit Test Cases Generation[C]. 2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST), 2018:432-433.

[19] Z. Chen and M. Monperrus. There markable role of similarity in redundancy-based program repair. arXiv preprint arXiv:1811.05703, 2018.

[20] 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 et al. Automated software vulnerability detection with machine learning. arXiv preprint arXiv:1803.04497, 2018.

[21] M. Allamanis, E. T. Barr, C. Bird, and C. Sutton. Suggesting accurate method and class names. In Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, pp. 38–49. ACM, 2015.

[22] M. Pradel and K. Sen. Deepbugs: a learning approach to name-based bug detection. Proceedings of the ACM on Programming Languages, 2(OOPSLA):147, 2018.

[23] 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), pp. 756–758. IEEE, 2016.

[24] K. Redmond, L. Luo, and Q. Zeng. A cross-architecture instruction embedding model for natural language processing-inspired binary code analysis. arXiv preprint arXiv:1812.09652, 2018.

[25] F. Zuo, X. Li,Z. Zhang,P. Young,L. Luo,and Q. Zeng. Neural machine translation inspired binary code similarity comparison beyond function pairs. arXiv preprint arXiv:1808.04706, 2018.

[26] M. Allamanis, M. Brockschmidt, and M. Khademi. Learning to represent programs with graphs. arXiv preprint arXiv:1711.00740, 2017.