CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Evaluating the Trade-offs of Diversity-Based Test Prioritization: An Experiment

Bachelor of Science Thesis in Software Engineering and Management

RANIM KHOJAH
CHI HONG CHAO

Department of Computer Science and Engineering
UNIVERSITY OF GOTHENBURG
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2020

CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

An experiment that compares Diversity-based Test Prioritization techniques in terms of coverage, detected failures and execution time, on different levels of testing.

A fractional factorial experiment that focuses on the evaluation of the trade-offs of artefact-based techniques namely, Jaccard, Levenstein, NCD and Semantic Similarity on unit, integration and system levels of testing.

Supervisor: FRANCISCO G. DE OLIVEIRA NETO
Examiner: Richard Berntsson Svensson

University of Gothenburg
Chalmers University of Technology
Department of Computer Science and Engineering
SE-412 96 Göteborg
Sweden
Telephone + 46 (0)31-772 1000

Department of Computer Science and Engineering
UNIVERSITY OF GOTHENBURG
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2020

Evaluating the Trade-offs of Diversity-Based Test Prioritization: An Experiment

Ranim Khojah
Department of Computer Science and Engineering
University of Gothenburg
Gothenburg, Sweden
guskhojra@student.gu.se

Chi Hong Chao
Department of Computer Science and Engineering
University of Gothenburg
Gothenburg, Sweden
guschaoch@student.gu.se

Abstract*—*Background: **Different test prioritization techniques detect faults at earlier stages of test execution. To this end, Diversity-based techniques (DBT) have been cost-effective by prioritizing the most dissimilar test cases to maintain effectiveness and coverage with lower resources at different stages of the software development life cycle, called levels of testing (LoT). Diversity is measured on static test specifications to convey how different test cases are from one another. However, there is little research on DBT applied to semantic similarities of words within tests. Moreover, diversity has been extensively studied within individual LoT (unit, integration and system), but the trade-offs of such techniques across different levels are not well understood.**

Objective and Methodology: **This paper aims to reveal relationships between DBT and the LoT, as well as to compare and evaluate the cost-effectiveness and coverage of different diversity measures, namely Jaccard's Index, Levenshtein, Normalized Compression Distance (NCD), and Semantic Similarity (SS). We perform an experiment on the test suites of 7 open source projects on the unit level, 1 industrial project on the integration level, and 4 industry projects on the system level (where one project is used on both system and integration levels).**

Results: **Our results show that SS increases test coverage for system-level tests, and the differences in failure detection rate of each diversity increase as more prioritised tests execute. In terms of execution time, we report that Jaccard is the fastest, whereas Levenshtein is the slowest and, in some cases, simply infeasible to run. In contrast, Levenshtein detects more failures on integration level, and Jaccard more on system level.**

Conclusion: **Future work can be done on SS to be implemented on code artefacts, as well as including other DBT in the comparison. Suspected test suite properties that seem to affect DBT performance can be investigated in greater detail.**

***Index Terms*—Diversity-based testing, Test Case Prioritization, Natural Language Processing (NLP), Level of Testing (LoT).**

I. Introduction

Testing is crucial in a software-intensive system to ensure a satisfactory degree of quality. Ideally, the entire test suite is executed on the System Under Test (SUT) to uncover failures, but in reality the increasing system complexity along with limited resources prohibit this. To achieve cost-effective testing within such conditions, test prioritization approaches aid testers to decide on what and how much to test. Several types of test case prioritization exist depending on prioritization criteria. Specifically, Similarity- or Diversity-Based Test case Prioritization has shown promising results and advantages for automated test optimization, being able to reduce test costs while keeping a satisfactory test coverage of the system [1], by measuring how different tests are from each other through distance functions for each pair of tests.

Testers intuitively assume that diverse tests result in a higher test coverage— the amount of functional requirements covered by a test— consequently probing more varied behaviours of the SUT. This in turn increases fault detection rate when testing is prohibitive [2], [3]. Predominantly, diversity is measured through distance functions that convey how different two pieces of information are from one another. Consequently, DBT require a concrete definition of the type of diverse information that is being measured, which can range from textual similarity [4], test input data [3] or test execution log patterns [5], [6]. DBT have shown to expose a similar amount of failures even without access to source code [5], [7]. Instead, testing artefacts or information sources from test cases executions are used.

Current research on diversity-based approaches present many strategies to measure diversity, each with their own contributions and limitations when it comes to applicability, performance and domain suitability. Moreover, the level of testing (unit, system and integration) should be considered alongside resource restrictions when choosing DBT. Levels of testing (LoT) are groupings of tests in different stages of the software development lifecycle where testing is performed. For instance, system-level tests are mostly written in natural language, enabling testers to verify and validate system features and user requirements, whereas unit tests are written in a programming language to examine a component at a lower level. In both cases, testers want to achieve diverse test coverage, but current research does not show how diversity measures perform on those different levels. Selecting a sub optimal DBT may drastically impact test prioritization performance.

One main type of Diversity-based techniques (DBT) is called Artefact-based diversity(a-div), which compares aspects of test specifications such as requirements, test inputs, or system output data to determine the similarity between tests. A subgroup of a-div compares distance between strings to illustrate how dissimilar two test cases are [4], [8]. The meaning of a word may change depending on the context and, currently, most of the existing string-based techniques mainly observe lexical, rather than semantic differences of test

cases regardless of the level of testing [4]. This may result in inaccurate test suite prioritization [6], reducing effectiveness. Therefore, Semantic Similarity (SS) is used in this experiment as an approach to measure the semantic distances between test cases. SS uses Natural Language Processing (NLP), a branch of artificial intelligence, to compare words, paragraphs, or documents to account for varying definitions of words.

In summary, we address two problems: (i) diversity measures are generic and applicable to any artefact [3], [5], however little work has been done on comparing DBT *across* several LoT in a holistic manner, and (ii) most string-based diversity measures do not capture semantics of test artefacts that are relevant for identifying relevant tests [2], [4].

In order to evaluate the trade offs of the four DBT, we need to be able to make sure that the techniques are the root cause of the observed differences. Thus, we perform a fractional factorial experiment to observe how different DBT perform on 7 open source projects and 4 projects from two industrial companies on three levels of testing. We compare coverage, failure detection rates and execution time of the techniques on the integration and system level, but only time and failure detection rates on the unit LoT. We measure coverage in terms of test requirements, which is feasible for system-level tests. However, we do not analyze coverage on the unit level due the conceptual differences between requirement coverage and code/conditional coverage, hence avoiding the analysis of disparate constructs. Through this experiment, we expect to contribute on both a technical and scientific perspective, namely:

- An experimental study that investigates the applicability, performance, and cost of several DBT on open source and industry data, on three levels of testing. Our implementations to obtain data for coverage, failure detection rates and execution times for Python Projects, Java projects and the Defects4J framework can be reused for future experimental studies for DBT.
- Our instrumented workflow can be adapted to be used for practitioners to run test case prioritisation techniques in their project's test suite.
- An implementation of Semantic Similarity [1] which makes use of Doc2Vec [9] and the Cosine Distance to rank a test suite. This can benefit future studies related to semantic string-based diversity or practitioners seeking to utilize such a technique under optimal conditions.
- Analysis of DBT trade-offs in regards to coverage, failure detection and execution time on three LoTs. An in-depth comparison between system and integration levels is presented. The results of SS are particularly novel.
- A list of recommendations of the optimal scenarios to use certain techniques based on our analysis results.

Our thesis is structured as follows: Section II highlights the research that are related to our experiment and explains some of the crucial concepts we present. Section III describes our experiment process and the steps we took to collect the data for the experiment. Section IV presents the results and the analysis of our experiment, and section V interprets the results with respect to our research questions. Section VI explores and discusses the different types of validity threats to our research and VII includes final insights and possible future work.

[1]Available at: https://github.com/ranimkhojah/Lemon-Ginger-Thesis

II. Background

A. Levels of Testing

Tests are usually grouped into specific "levels" to make tests systematic and focus on a certain purpose and aspect of a software while testing it. In this experiment, we focus on three levels of testing: unit, integration, and system levels. Testing on a unit level examines each component of a SUT independently and ensures that it returns the expected outcome.

Unit testing focuses on checking if an isolated unit of a system behaves as expected. The unit tests focused on in this experiment are JUnit tests written in Java where unit that is being tested is a method in a class. Listing 1 is an example of class methods that are tested using the test suite in Listing 2.

Integration-level testing on the other hand concerns itself with the dependencies between different parts of the software and ensures that they are compatible together. We focus on testing the dependencies between entries of the project's modules e.g. API endpoints and other classes of the project. Example of an integration test of API endpoint is illustrated in Listing 4.

Finally, the software as a whole is tested on a system level to ensure that the software fulfills the user requirements and system features. System tests can be written as code or in natural language, where the latter will be used in this experiment. As shown in Listing 3 and the documentation part in Listing 4, system tests will be represented by test case descriptions and/or test specifications.

Different companies do testing at different levels (unit, integration and system), and comparing benefits across those levels is particularly challenging since each explore a unique test purpose. However, the diversity measures are, in theory, applicable to any type of artefact [5]. Therefore, our goal is to see whether diversity can also be captured and prioritised across those different levels of testing.

B. Test Diversity - An Example

We illustrate the appeal of DBT with a toy example where our SUT is the class MyFarm (Listing 1), along with the corresponding unit (Listing 2) and system tests (Listing 3). Consider the unit and system test suites, each containing 7 test cases. We can see that testEggNum() and testIsEggEmpty() are similar. Likewise, testMilkNum() is similar to testIsMilkEmpty(). On the system level, one can easily see which scenarios are related to eggs ["Number of Eggs", "Egg Status"] or milk ["Number of Milk", "Milk Left in Farm"].

Given that there is only enough resources to execute 3 of 6 tests on each level, our goal would be to still cover all features with 3 tests for both levels. While there is no one right answer, a valid answer could be to run [getChickens(), getMilkNum(), isEggEmpty()] on the unit level, and [Get Number of Cows,

Egg Status, Milk Left in Farm]. These three test cases would still maintain the breadth of coverage as all features would be covered as much as possible. It should be noted that tests of different LoTs are not ranked together. While this example SUT has a system test for each unit method, in reality a system test examines multiple code components together.

```
public class MyFarm {
 private int chickens, eggNum, cows, milkNum;

 public MyFarm (int chickens, int cows) {
   this.chickens = chickens; this.cows = cows;
   this.eggNum = 5;          this.milkNum = 10;}

 public int getChickens(){ return this.chickens;}
 public int getCows()    { return this.cows;}
 public int getEggNum()  { return this.eggNum;}
 public int getMilkNum() { return this.milkNum;}
 public boolean isEggEmpty() {return eggNum ==0;}
 public boolean isMilkEmpty(){return milkNum==0;}}
```

Listing 1. Our class under test is a farm with animals.

```
public class MyFarmTest {
  private static int CHICKENS, COWS = 5;
  private static int EGGCOUNT = 5;
  private static int MILKCOUNT = 10;
  private MyFarm farm;

  @Before public void setUp()
    {farm = new MyFarm(CHICKENS, COWS);}
  @Test public void testChickens()
    { assertEquals(CHICKENS, farm.getChickens());}
  @Test public void testCows()
    { assertEquals(COWS, farm.getCows());        }
  @Test public void testEggNum()
    { assertEquals(EGGCOUNT, farm.getEggNum());  }
  @Test public void testMilkNum()
    { assertEquals(MILKCOUNT, farm.getMilkNum());}
  @Test public void testIsEggEmpty()
    { assertFalse(farm.isEggEmpty());            }
  @Test public void testIsMilkEmpty()
    { assertFalse(farm.isMilkEmpty());          }}
```

Listing 2. Example of unit tests to cover the class under test.

```
Scenario: Get Chicken Number
   Given there are 5 chickens in the farm
   When the user queries the chicken amount
   Then the 5 chickens should appear in the coop
Scenario: Obtain Number of Cows
   Given there are 5 cows in the farm
   When I check the remaining cows in the farm
   Then the 5 cows should appear in the farm
Scenario: Egg Quantity
   Given there are 5 eggs left in the farm
   When the farmer checks how many eggs are left
   Then the farmer should see 5 eggs are left
Scenario: Number of Milk
   Given there exists 10 milk
   When I investigate how much milk is left
   Then I should see 10 milk left in the farm
Scenario: Egg Status
   Given the farm has no more eggs
   When the farmer considers if the farm has eggs
   Then the farm should show that no eggs exist
Scenario: Milk Left in Farm
   Given there is more than 1 milk in the farm
   If I check the status of the milk
   Then I should see that milk exists in the farm
```

Listing 3. Example of system tests to cover the class under test.

Using a string-based diversity test prioritization technique can automatically determine which tests to run under such circumstances, but reality is often more complex. As string-based techniques only look at the lexical aspect of individual words, context is not taken into account. This could result in incorrect test prioritization, such as having both [”Number of Eggs”, ”Egg Status”] system level tests being chosen instead of a combination between Eggs and Milk. Factors such as different test authors, or synonyms in different tests can make the string-based technique ”think” that ”Egg Status” was more diverse than, e.g., ”Milk Left in Farm”. SS, on the other hand, would likely spot such semantic differences and determine that ”Egg Status” and ”Number of Milk” should be prioritized first.

There may be a point of diminishing returns where it may not be needed to run more expensive techniques to acquire a more optimal prioritization. For instance, running [”Number of Eggs”, ”Egg Status”] still covers a large majority of features, and perhaps it is enough to simply use a faster, but less effective DBT. This is especially true in this toy example, as the features are similar in implementation (isEggEmpty() and isMilkEmpty() are nearly identical). However, a realistic system can contain much more important, nuanced, and complex differences that SS may spot in contrast to lexical string-based techniques. These are the trade-offs between techniques that this experiment attempts to shed more light on.

C. Diversity-based Prioritization

Diversity-Based Test Case Prioritization has contributed to automated test optimization by enhancing the coverage at a low cost [1], and supporting data-driven decision making on test maintenance [2]. Studies have also shown that diversity-based selection performs better in detecting faults with fewer test cases compared to, e.g., manual selection, especially if the test suite has a medium or high amount of redundancy in test cases [3], [10]–[13].

DBT require some definition of what type of diverse information is being measured, such as the diversity of system requirements [1], [11], code statements, execution logs [5], [6], or test steps [2]. There are a multitude of techniques which have unique benefits and drawbacks that come from various aspects. Diversity can be measured using textual similarity [4] or general diversity between objects [5], for example. The level of tests that are required can be different - tests covering diverse requirements [1], [2], test input and output [5] or even test scenarios [11] can be used. Normalized Compression Distance (NCD), for example, calculates diversity by measuring how difficult it is to transform any 2 objects into each other, but is generally more computationally expensive [3]. In our experiment, we focus on evaluating techniques that capture similarities by following the process defined in Fig. 1.

After mining test repositories, tests are encoded into vectors (if the technique requires it) in order to measure pairwise distances between test cases. Next, the encoded test information is given to a distance function, resulting in a distance value. When the distance values are normalized, a pair of test cases are considered to be identical if the distance between them

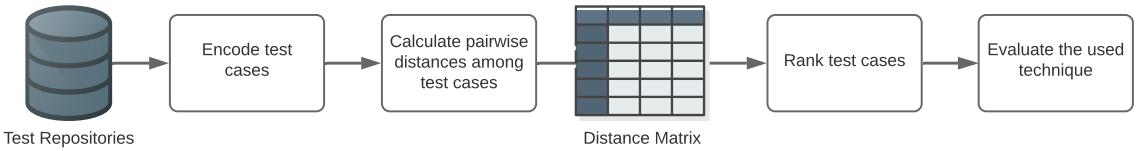

Fig. 1. The workflow of the experiment. This figure is partially based on de Oliveira Neto et al.'s [2] illustration of diversity-based test optimisation steps.

is 0, and totally dissimilar if the distance between them is 1. These pairwise distance values are then arranged in a matrix, which is used for other testing activities.

Considering $T = t_1, t_2, ..., t_n$, a distance matrix D is an nxn matrix, where $n = |T|$, that is, the length of the test suite. D includes the pairwise distances between test cases, for instance, the value of $D(t_i, t_j)$ is the distance between test case t_i and test case t_j. This experiment uses different techniques that read and encode test information, measure distances and create a distance matrix accordingly. The techniques that were used, namely, Jaccard's Index, Levenshtein, NCD, and SS, are summarized in Table I. Next, we detail our usage of each of the DBT.

1) Jaccard's Index: Jaccard's Index [14] is used to extract the tests information and breaking them down into sequences of n characters called n-grams. Subsequently, it measures the lexical similarity based on how much test cases have characters in common, in other words, how many n-grams the test cases share. Accordingly, the Jaccard distance i.e. dissimilarity between a pair of test cases t_i and t_j is measured through Equation 1.

$$jaccardDistance(t_i, t_j) = 1 - \frac{|t_i \cap t_j|}{|t_i \cup t_j|} \quad (1)$$

2) Levenshtein: Levenshtein defines the distance between t_i and t_j as the minimal number of operations e.g. insertion, deletion, replacement to change t_i into t_j. for instance, the distance between "tree" as S1 and "bee" as S2 is 2, where S1 needs one deletion of letter "t" and one replacement of letter "r" with "b" in order to transform to S2. Levenshtein can be calculated using Equation 2 where ti and tj are the lengths of t_i and t_j respectively.

$$Lev(ti, tj) = \begin{cases} max(ti, tj) & if min(ti, tj) = 0, \\ min \begin{cases} Lev(ti-1, tj) + 1 \\ Lev(ti, tj-1) + 1 \\ Lev(ti-1, tj-1) + 1_{ti \neq tj} \end{cases} & otherwise, \end{cases} \quad (2)$$

3) Normalized Compression Distance (NCD): NCD similarity [15] between two documents x and y assumes that the if the concatenation xy of x and y was passed to a compressor C, then the compression ratio is the similarity between x and y, hence 1- the similarity is the NCD distance between x and y which can be calculated by Equation 3.

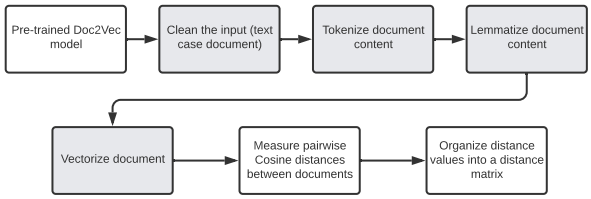

Fig. 2. The steps to measure semantic similarities between test cases. The grey boxes indicate the phases related to the NLP-approach.

$$NCDdistance(x, y) = 1 - \frac{C(xy) - min(C(x), C(y))}{max(C(x), C(y))} \quad (3)$$

4) Semantic Similarity (SS): We made use of NLP in this experiment to capture semantic similarities between documents by following the steps specified in Fig. 2.

We capture semantic similarities between test cases using Doc2Vec or Paragraph Vector which is an unsupervised framework introduced by Le and Mikolov [9] to capture features of the document content in a vector with respect to the words' semantics and ordering in a paragraph. More specifically, we use a pre-trained Doc2Vec model on Wikipedia data[2], which we believe covers the knowledge required to get our SS model to understand natural language.

The test case description documents are the main artefact that SS extracts test information from, these documents come from the system-level tests specifications that describe the main purpose of a given test case along with the conditions, steps and the expected outcome. So, when the test specifications (including several test case descriptions documents) are passed to the SS pipeline, a cleaning process is performed on the documents to remove non-Latin characters, non-English words, URLs, punctuation and stop words, i.e., the most common words in a language such as pronouns or conjunctions. Then, the content is tokenized into words in order to perform lemmatization that converts each token to its root, e.g. verbs "to be" are converted to "be" and verbs in a specific tense are lemmatized to the infinitive tense.

Finally, Doc2Vec uses the Paragraph Vector algorithm to construct a vector representation of each document. Moreover, Doc2Vec has a built-in function to compute the Cosine distance [16], [17] that we use to measure the pairwise distances

[2]https://github.com/RaRe-Technologies/gensim

TABLE I
SUMMARY OF THE PRIORITIZATION TECHNIQUES USED IN THIS EXPERIMENT

	Description	Advantages	Disadvantages
Jaccard's Index	Captures lexical similarity by checking the commonalities between two strings based on substrings of a string (q-grams).	1. Simple to interpret, fast to execute. 2. Gives positive results in large datasets and usually used as a baseline in literature.	1. Limited to the intersection between two strings when measuring distance. 2. Sensitive, erroneous in small datasets.
Levenshtein	Defines distance between two strings as the number of edit operations required to transform the first string to the other.	1. Theory is simple to understand. 2. Efficient for short strings.	1. Computationally expensive. 2. Inefficient for long strings.
NCD	Compares two compressed strings with the compressed concatenation of these strings to measure the distance between them.	1. Doesn't need parameters and usable in any type of data (e.g., files, strings). 2. Robust to errors in feature selection.	1. Computationally expensive. 2. Compressor selection might be crucial to effectiveness.
Semantic Similarity	NLP-approach to extract features from test case specifications and creates vector representations for each document then measures pairwise document similarity using the cosine similarity function.	1. Captures semantic similarities with respect to words' order. 2. Cheap, vectors are learned from unlabeled data. 3. Flexible, can use any similarity function.	1. Training a model can be time consuming. 2. Very sensitive to the used model and the number of epochs during training.

among all test cases in a test suite, and then arrange them in a distance matrix. The Cosine distance computes the distance between two vectors A and B by measuring the cosine of the angle between them using Equation 4, where $A.B$ is the dot product between the two vectors.

$$CosineDistance(A, B) = 1 - \frac{A.B}{||A|| \times ||B||} \tag{4}$$

D. Related Work

Different areas of optimization emerged to reduce testing resources without hindering effectiveness, such as test case selection, prioritization and minimization [18]. Test case minimization tries to remove redundant tests, test case selection looks for test cases that are relevant to recent changes, and prioritization orders or ranks test cases such that faults can be detected earlier. While we focus on prioritization techniques, note that test case prioritization can be combined with test case selection and minimization to suit specific contexts.

Many studies have looked into test case prioritization. Yoo and Harman [18] surveyed and analyzed trends in regression test case selection, minimization and prioritization. They found out that these topics are closely related and reported that the trends suggest test case prioritization had increasing importance, and that researchers were moving towards the assessment of complex trade-offs between different concerns such as cost and value, or the availability of certain resources, such as source code. To this end, Henard et al. [7] experimentally compared white box and black box test prioritization techniques, and found that diversity based techniques, along with Combinatorial Interaction Testing, performed best in black box testing. They also found a high amount of fault overlap between white and black box techniques, indicating that an acceptable amount of faults can still be uncovered even without source code available.

While Henard et al. revealed that diversity based techniques managed to find an acceptable amount of faults with restricted resources [7], de Oliveira Neto et al. expanded on that and found that the visualization of the same diversity information helped practitioners in test maintenance and decision making as well [2], indicating that the benefits of diversity based techniques are multifaceted, depending on the context and the usage.

Hemmati et al. [19], [20] conducted a case study as well as a large scale simulation to look into how test suite properties of model-based testing affected diversity-based test case selection, and found that such diversity techniques worked best when test cases that detect distinct faults are dissimilar, and not so well when many outliers exist in a test suite. In response, Hemmati et al. introduced a rank scaling system, which partially alleviated the problem.

In turn, Feldt et al. [5] presented a model for a family of universal, practical test diversity metrics. One subset of techniques compare string distances in order to measure diversity. Strings are compared lexicographically and a string distance is given to illustrate how dissimilar two test cases are. de Oliveira Neto et al. used Jaccard's Index, one of such techniques, to visualize company test cases to trigger insightful discussions [2]. However, most of the techniques are unable to capture semantic similarities while comparing test cases regardless of the level of testing. This may result in inaccurate test suite prioritization since tests that semantically related features may not be detected by simply comparing strings (e.g., braking and acceleration features in automotive components) [1], [6].

Although rare, capturing semantic similarities in the comparison between test cases has been attempted. Tahvili et al. [21] presented a NLP approach that revealed dependencies between requirements specification, and performed a case study on an industrial project. They suggested the dependency information can be utilized for test case prioritization, and found that using NLP on a integration level of testing is feasible. Yet, the paper only compared NLP with Random prioritization and did not include common string-based distances such as Jaccard used in other diversity-based studies. This is problematic as the comparison between NLP and Random is unbalanced and rather partial towards NLP.

TABLE II
SCOPE AND VARIABLES OF OUR EXPERIMENTAL STUDY.

Objective	Explore
Experimental Design:	Fractional Factorial Experiment
Experimental Units:	Unit tests, integration tests and test specifications.
Experimental Subjects:	4 industrial test suites 7 open-source projects
Dependent Variables:	Coverage, detected failures and execution time.
Factors:	Technique (F1), Levels of testing (F2)
Levels for F1:	Jaccard, Levenshtein, NCD, SS and Random
Levels for F2:	System, integration and unit level
Parameters:	Programming language, test suite size.

III. METHODOLOGY

The primary research method for this study is an experiment that is designed to evaluate the trade-offs of diversity measures on different LoT. We focus on three main test levels, i.e. Unit, Integration and System LoTs. Unit-level test artefacts considered here are tests written in a xUnit framework (e.g. JUnit) that test a class. We use integration tests that have function calls to entries of the SUT's modules (e.g., API endpoints). System-level tests are written in natural language and describe the user actions and expected systems output. Regarding diversity measures, SS is only applied to system-level artefacts, because it is not applicable to programming languages. String distances however, are used on all LoT. Since some of the treatments (i.e., combination of levels between factors) are not be feasible, we use a fractional factorial experimental design. We aim to answer the following research questions:

RQ1: How do DBT perform in terms of coverage on the system and integration levels?
RQ2: To what extent does each DBT uncover failures?
RQ3: How long does it take to execute each technique on different level of testing?
RQ4: How do different levels of testing affect the diversity of a test suite?

The experiment executes each technique on certain levels of testing following the process defined in Fig. 1. The techniques and LoTs are the independent variables, while we compare the following dependent variables: coverage, execution time, and failure detection rate. We also run a Random test prioritisation as a baseline, which only looks at the names of the tests, then shuffles them into a list as a prioritized test suite. Random is executed on time, coverage and failures 100 times and then their results are averaged. Table II summarises the components of our experiment.

Diversity measures rely on the content of tests to determine distance values. Note that different diversity measures evaluate different parts of the artefact, for instance, Levenshtein preserves sequences of characters, whereas NCD is generic to any type of file. Therefore, we aim to evaluate whether those differences affect the diversity of 11 test suites in total

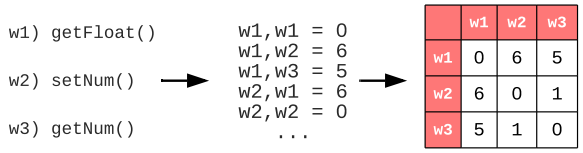

Fig. 3. An example of a distance matrix generated based on the pairwise distances between the strings: `getFloat()`, `getNum()` and `setNum()`, using Levenshtein distance function. Note how w2 and w3 are perceived as very similar to each other.

and, consequently, our dependent variables. Therefore, we ran diversity techniques on test artefacts that represent test content differently, e.g., test steps via code statements (unit), function calls (integration) and step descriptions (system). In detail, We ran the string distances using the MultiDistances package[3] which offers an implementation in Julia[4] of various a-div techniques and has been used in previous studies [2], [3], [6]. The package reads a test suite as a directory that contains test cases files in different formats such as text documents (.txt), JUnit (.java) or XML exported from life-cycle management systems. Then, it creates a distance matrix as illustrated in the example in Fig. 3. The MultiDistances package also ranks the test suite with regards to the generated distance matrix using the Maximum mean distance between tests, such that tests that have the higher distance value are ranked higher than very similar tests (i.e., low distance values). It then checks the second highest distance value and performs the same actions, until all test cases are ranked.

We also instrument a tool for Semantic Similarity (SS) with an NLP-based approach using an off-the-shelf Doc2Vec model. The implementation is done in Python[5] and it follows the steps defined in Fig. 2. SS can either read test specifications as a directory that contains tests descriptions as individual text documents, or it can extract test descriptions written as documentation in a test function (Example in Listing 4). The test specification artefacts used in this experiment describe a test case in 3 levels of detail since it consists of the test name, steps and expected outcome.

```
import requests
Tested_Requirement ="GetCows"
def test_get_cows_from_api():
    """
    Test: Get all cows from myfarm API
    Expected Outcome: "200 OK" HTTP status code
    Steps:
        1. Send get cows request to cows endpoint
        2. Verify that the HTTP status is OK
    """
   response = requests.get('http://myfarm.se/cows')
   assert_true(response.ok)
```

Listing 4. The structure of an integration test that includes a system-level test written as documentation

Semantic Similarity (SS) makes use of Doc2Vec to vectorize documents [16], [22] by capturing string features of document

[3]https://github.com/robertfeldt/MultiDistances.jl
[4]https://julialang.org/
[5]https://www.python.org/

TABLE III
THE SYSTEMS UNDER TEST USED IN THIS EXPERIMENT. A. = AVERAGE NUMBER OF TEST CASES. M. = MEDIAN OF THE TEST CASES NUMBER.

SUT	Description	Source	LoT
Project 1	2639 TCs (test specifications)	CompanyA	System
Project 2	875 TCs (test specifications)	CompanyA	System
Project 3	2691 TCs (test specifications)	CompanyA	System
Project 4	1605 TCs (test specifications and integration tests)	CompanyB	System/ Integration
Cli	A. 262, M. 248 TCs (39 faults)	OpenSource	Unit
Codec	A. 440, M. 344.5 TCs (18 faults)	OpenSource	Unit
Gson	A. 988, M. 994.5 TCs (18 faults)	OpenSource	Unit
JacksonCore	A. 356, M. 344 TCs (26 faults)	OpenSource	Unit
JxPath	A. 347, M. 342 TCs (22 faults)	OpenSource	Unit
Lang	A. 1786, M. 1716.5 TCs (64 faults)	OpenSource	Unit
Math	A. 2513, M. 2319 TCs (106 faults)	OpenSource	Unit

and represent these features by generating a vector that corresponds to a document (a test case in a test suite). Based on a provided corpus (Wikipedia data), we use a pre-trained Doc2Vec model to perform text-similarity tasks and to calculate the pairwise distances between the generated vectors using the Cosine Distance. Lastly, we arrange all pairwise distances in a distance matrix. In order to ensure consistency, Maximum mean is also performed using MultiDistances package to rank the the test cases documents based on the distance matrix.

A. Data Collection

We collect data from two industry partners, and open source projects in GitHub (Table III). The two partners (Company A and B) vary in domain as the former is an IT sector of a retail company and the latter is a surveillance company. Company B provides test suites that contain integration and system tests, whereas Company A only provides system tests. In addition, we use open source data from Defects4J [6] [23], which provides unit tests that detect isolated faults along with specific information regarding tests that trigger such faults.

We measure coverage by using the traceability information of each test on the integration and system LoT and its corresponding requirement. As there are no requirements at the unit level, **Coverage is not measured at the unit level**. Failure detection rate is measured in terms of the Average Percentage of Failures Detected (APFD) [24], i.e., how early the prioritized test suite detect failures. Finally, as DBT are usually inefficient when performing a large number of pairwise comparisons [3], [13], [25], we considered the time required to perform the prioritization–including the distance matrices generation–to help addressing a bigger picture of the trade-off that each technique presents. Although we need to adjust data collection to each LoT, note that the same metrics are used amongst the levels of testing (with a few exceptions detailed below). This allows us to address RQ4 and compare those different levels based on the findings from each technique's assessment.

[6]https://github.com/rjust/defects4j

1) Unit-level Data: The D4J framework was selected due to its large collection of real, reproducible faults, each with documented properties and triggering tests. A total of seven open source projects were used as test subjects on the Defects4J (D4J) framework. Although there is a total of 17 projects in D4J at the time of writing, early technical issues and later time constraints prevented us from using all 17. Despite these issues, We still wanted a range of projects of varying sizes, both in terms of number of tests and byte size. The seven projects were thus selected due to convenience and differences in size. For each D4J project fault, there are two unique project versions - a "faulty" (buggy) version that contains the isolated fault, and a "fixed" version that removes the fault. Note that since the project's faults are found across different releases of the SUT, each faulty/fixed versions contain a different test suite (as both the system and test suite evolved). This meant that the size and contents of each version is different, and versions from different faults could not be merged into a single version with many faults. For consistency, only the fixed versions were used in this experiment.

The steps to execute the experiment on the Unit LoT are as follows: 1) Obtain all the fixed versions for all faults in a project, 2) For each version's test suite, extract each test method and which triggers the fault, 3) Calculate time and failures for prioritising each test suite version separately, and aggregate (mean) the results for each project. We calculate the failures revealed at different budget cutoffs (i.e., the APFD). In other words, how many failures would be revealed by only executing a portion (e.g. 30%) of the tests. To be consistent with other LoTs—which do not have fault information available—we count the total of failures, instead of faults.

2) Integration-level Data: Data was gathered on integration-level from Project 4 that included 1605 tests. Each integration test could be traced to a single system-level test as well as a single requirement (See Listing 4). Project 4 is also supported by the failure information of the integration tests over 669 builds. The artefact consisted of the *test steps* along with the *expected outcome* that include detailed information regarding the elements that the test case covers.

In this experiment, we focus on requirements coverage which is satisfied when the test suite contains at least one test that is mapped to at least one system requirement of the SUT [26]. In Project 4, the integration tests were extracted then linked to a requirement using Algorithm 1, that produced a list of all integration tests with the corresponding system test and requirement.

Then given a list of the linked tests, the ranking of the prioritized test suites is used to determine how early the respective test suite covers a new requirement by adding a flag to each test case which tells whether the test case has tested a new requirement or not in Algorithm 2.

On the other hand, the failure information available for project 4 contained failures for different builds and test executions. We filtered the execution history to include only builds

Algorithm 1: Extract test artefacts in Project 4

```
while there are functions to read do
    if the function is an integration test then
        read the integration test and test description;
        create a link between the two artefacts;
    end
end
```

Algorithm 2: Record Requirement Coverage

```
visitedReqs;
for each TC in the ranked test suite do
    if the TC's requirement not in visitedReqs then
        report that the TC "covers a new requirement";
        add the requirement to visitedReqs;
    end
end
```

that contained at least one failure. Furthermore, 115 out of 669 builds were used in this project, and the relevant failure information regarding the test cases' names and result for the respective builds were collected.

3) System-level Data: The data gathered on a system-level was obtained from Projects 1,2,3 and 4. Projects 1-3 are provided by Company A and include system-level test specifications, the test specifications are written by testers that have good knowledge about the SUT. In addition, most of the test case specification consist of the test steps along with the corresponding expected outcome from the SUT. However, since the test specifications are written by human testers, there are many test case specifications that either don't follow a standard (e.g., have missing expected outputs, or incomplete actions) or are duplicates of other test case specification. In contrast, Project 4 includes a test suite with system-level test specifications (as in Listing 4) that are mined and extracted by a tool that follows Algorithm 1.

Requirement coverage information was collected for projects 1-4 using the same method explained under *Integration-level Data* and shown in Algorithm 1. Then We build maps between test cases and corresponding linked requirements to record coverage using Algorithm 2.

Moreover, the failure information was provided by only Company B. Therefore, failure detection rate was measured only for Project 4.

4) Measuring time efficiency: Last but not least, the efficiency of the DBT on all LoT is represented by the wall-clock time taken for each technique to fully execute. The techniques were timed by the Unix time utility when executed in two virtual machines with 4GB RAM each, and using two computers: a MacBook Pro, with 3 GHz Intel Core i7 and 16 GB RAM, along with a Lenovo Legion Y530, with a 2.2 Ghz Intel Core i7 and 32 GB RAM.

All techniques on system and integration level were executed **10 times** per project to account for Maximum mean randomness. The Maximum Mean algorithm implements some random decisions when deciding which test case to prioritize and which test case to deprioritize when two test cases have the shortest distance between them, meaning that they are very similar. On the unit level, techniques were executed **once** per version due to high cost. For example, executing NCD on one of the project's versions (Lang) took an average of 12 minutes. Multiplied by each version (64 faulty/fixed versions, see Table III), the total execution time was 12 hours. Running the same technique five times would take 2.5 days, which was too costly. Nevertheless, Random was executed 100 times since it was cheap to run.

IV. Experimental Results and Analysis

Here, we present and describe data and results for each of our RQ, along with summarized answers. In turn, we discuss the reasons and insights drawn from our results in Section V. All the statistical analyses below followed the empirical guidelines on software engineering in research [27].

A. How do diversity-based techniques differ on system and integration levels in terms of coverage?

We executed all techniques at system level on 4 projects and then obtained coverage percentage for each project (presented in Fig. 4). The plots illustrate the percentage of covered feature requirements for a given number of test cases (Budget) using different techniques on integration and system level.

For system-level coverage for the projects provided by Company A (i.e. Project 1-3), all the techniques took a linear shape which indicates the mediocre performance and feature coverage. However, a different behaviour is revealed in the project provided by Company B (i.e. Project 4) where SS took the lead by covering most features across a-div techniques. Surprisingly, Random was slightly better than NCD and Levenshtein in Project4.

For Integration-Level Coverage, as SS was not executed on integration level, Random showed the best performance across all techniques. Jaccard, Levenshtein and NCD had close performance until budget reaches ~30%. When the budget exceeds 30% Jaccard separates and shows a lower coverage than Levenshtein and NCD.

RQ1: SS performs best on system-level. On both LoTs, NCD and Levenshtein's coverage are similar. Jaccard covers the least features in all projects.

B. How do diversity-based techniques differ in terms of failure detection on different levels of testing?

We highlight visual differences between failure detection rate of each technique in our charts, then we verify our observations by performing a post hoc analysis that includes a Friedman's statistical test on all techniques to determine whether a statistical significant difference (SSD) exists. We use a Bonferroni correction for the pairwise post hoc test of our data using Wilcoxon Signed Ranked test. We measure effect

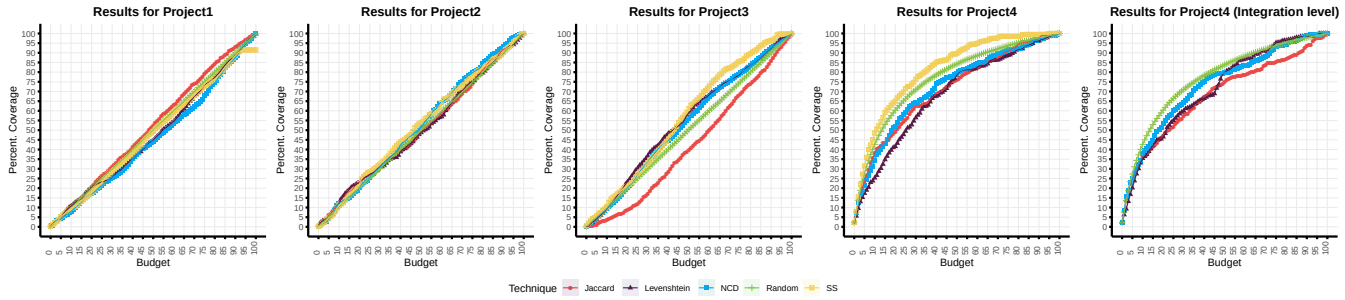

Fig. 4. The plots show the percentage of coverage for Projects 1-4. Budget values represent percentage of prioritized tests executed. Note that SS doesn't reach 100% coverage in Project1 since it ignored some empty files (test specifications) which were linked to some features.

size via Kendall's W to judge the effect level of the statistical differences between each pair of techniques (S- Small, M- Moderate, L- Large) . For simplicity, we chose three budgets to test SSD in the APFD in order to represent more prohibitive (30% test suite size), reasonable (50%) or permissive (80%) constrained testing scenarios.

1) Failures on Unit-level: On the unit level, all open source projects' results are presented in Fig. 5. Across all projects, there is a general trend of all DBT revealing more faults than Random, but a visual analysis does not show a clear pattern. Furthermore, it is difficult to see which of the techniques fare better than the rest.

According to the post hoc analysis presented in Table IV, the statistical tests confirm that there is indeed a significant difference, albeit small, for all pairs of techniques on the 30% budget. At the 50% budget, the effect sizes of all random pairs grow larger, as shown in the Kendall's W values, but there is a reduced difference between the DBT. This trend continues in the 80% budget, with all techniques having a large effect size compared with Random, but the a-div techniques have a smaller effect size amongst themselves, with Lev-Jaccard and Jaccard-NCD ceasing to have significant differences, supported by the small effect size.

2) Failures on Integration-level: Based on Fig. 6, the failure detection rate of the techniques is similar for small test budgets. However, the differences between the techniques start to be clearer after using 30% budget of the test suite. Finally, with a higher budget than 60% Levenshtein and NCD perform similarly the best whereas Jaccard falls to reach a failure detection rate lower than Random. On the other hand, the post hoc statistical analysis reported that Levenshtein on 30% budget was significantly different all techniques with a moderate effect size, whereas random/NCD and random/Levenshtein comparisons were not significantly different. At 50% and 80% budget, all pairwise comparisons are significantly different, and their effect sizes increase in general. However, at 80% budget, the statistical analysis reports that Levenshtein and NCD are significantly different. Even though there was a SSD, the effect size is small, which is also confirmed by the overlap of the curves in Fig. 6.

3) Failures on System-level: On a system level, Project 4 was the only one with available failure data. Based on Fig. 6 (left), we can see that SS was the closest to Random's performance across all techniques, followed by Jaccard. Furthermore, Levenhtein and NCD had a high and similar failure detection rate.

TABLE IV
SUMMARY OF THE POST HOC ANALYSIS ON DETECTED FAILURES ON UNIT LEVEL WHERE EACH ROW REPRESENTS A PAIRWISE COMPARISON.

comp.	p value	Adj.p val	Kendall's W	Eff. Size	SSD
30% Budget					
Rand-Lev	0.0001	<0.001	0.0021	S	Yes
Rand-NCD	<0.001	<0.001	0.0446	S	Yes
Rand-Jacc	<0.001	<0.001	0.0885	S	Yes
Lev-NCD	1.97E-08	<0.001	0.0471	S	Yes
Lev-Jacc	1.07E-14	<0.001	0.0558	S	Yes
NCD-Jacc	0.0343	0.034	0.0051	S	Yes
50% Budget					
Rand-Lev	<0.001	<0.001	0.2146	S	Yes
Rand-Jacc	<0.001	<0.001	0.2296	S	Yes
Rand-NCD	<0.001	<0.001	0.3893	M	Yes
Lev-Jacc	0.2101	>0.999	0.0042	S	No
Lev-NCD	2.66E-08	1.60E-07	0.0402	S	Yes
Jacc-NCD	1.64E-05	9.84E-05	0.0326	S	Yes
80% Budget					
Rand-Lev	<0.001	<0.001	0.6964	L	Yes
Rand-Jacc	<0.001	<0.001	0.7635	L	Yes
Rand-NCD	<0.001	<0.001	0.8680	L	Yes
Lev-Jacc	0.1061	0.6365	0.0167	S	No
Lev-NCD	0.0203	0.1220	0.0062	S	Yes
Jacc-NCD	0.4814	>0.999	0.0009	S	No

On the other hand, the post hoc statistical analysis revealed that at 30% budget SS is not significantly different from Jaccard and Levenshtein, whereas all other comparisons are significantly different but with a small effect size. At 50% Jaccard and SS remain significantly different with a small effect size whereas NCD's effect size increase to "Moderate" when compared with Jaccard and SS. At 80%, all the comparisons that include Random are significantly different, unlike SS which loses the SSD with other techniques. In addition, Jaccard becomes clearly different than other string distances (other than SS).

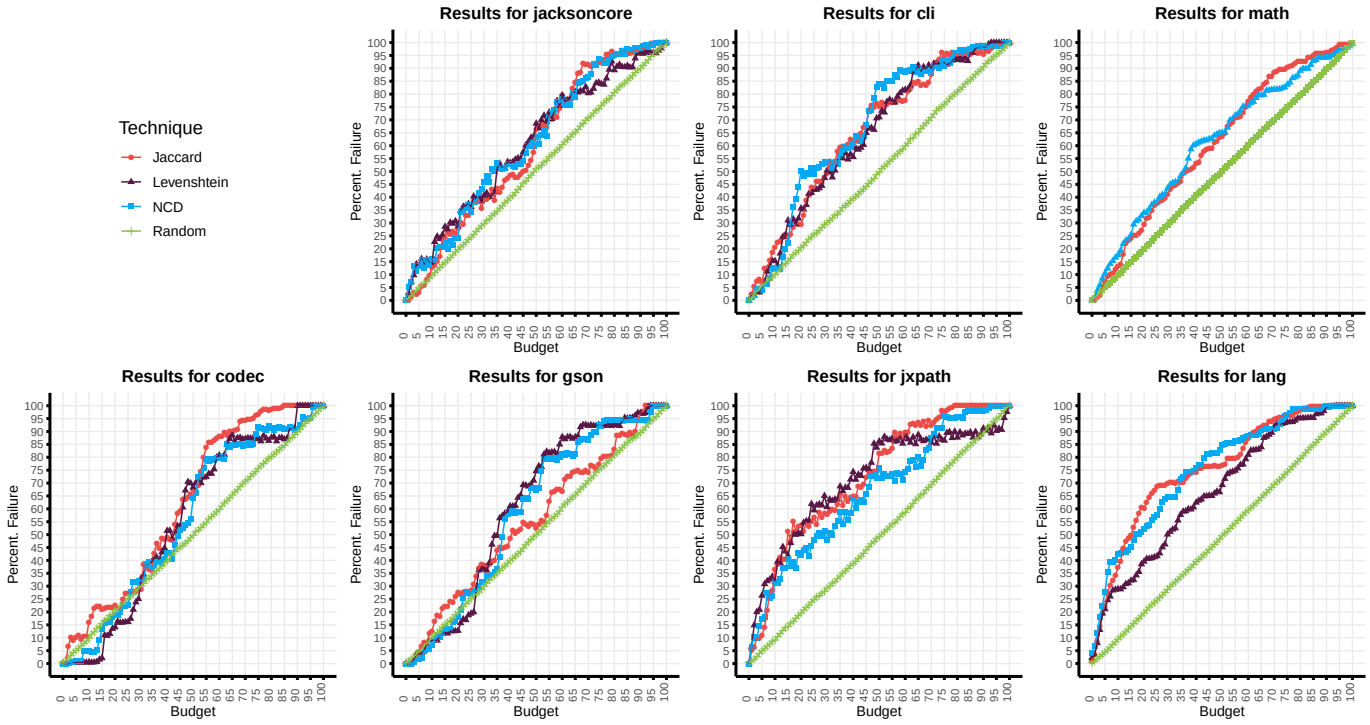

Fig. 5. Percentage of Failures found for all open source projects studied. Budget values represent percentage of prioritized tests executed.

TABLE V
SUMMARY OF THE POST HOC ANALYSIS ON DETECTED FAILURES ON INTEGRATION LEVEL.

comp.	p value	Adj.p val	Kendall's W	Eff. Size	SSD
		30% Budget			
Rand-Jacc	0.1629	0.9774	0.0122	S	No
Rand-NCD	0.2449	>0.999	0.0007	S	No
Rand-Lev	<0.001	<0.001	0.4793	M	Yes
Jacc-NCD	0.0105	0.06313	0.013	S	Yes
Jacc-Lev	<0.001	<0.001	0.4066	M	Yes
NCD-Lev	<0.001	<0.001	0.4281	M	Yes
		50% Budget			
Rand-Jacc	6.93E-11	4.16E-10	0.0003	S	Yes
Rand-NCD	<0.001	<0.001	0.477757	M	Yes
Rand-Lev	<0.001	<0.001	>0.999	L	Yes
Jacc-NCD	<0.001	<0.001	0.2845	S	Yes
Jacc-Lev	<0.001	<0.001	0.5994	L	Yes
NCD-Lev	<0.001	<0.001	0.2523	S	Yes
		80% Budget			
Rand-Jacc	<0.001	<0.001	0.0321	S	Yes
Rand-NCD	<0.001	<0.001	0.9024	L	Yes
Rand-Lev	<0.001	<0.001	0.9491	L	Yes
Jacc-NCD	<0.001	<0.001	0.3464	M	Yes
Jacc-Lev	<0.001	<0.001	0.4269	M	Yes
NCD-Lev	0.0003	0.0015	0.1026	S	Yes

RQ2: No technique consistently finds most faults on the 3 LoTs. However, there is a pattern where there is a greater distinction between DBT and Random as budget increases up until 80%. Statistically speaking, SS has SSD with the DBT, except with Jaccard.

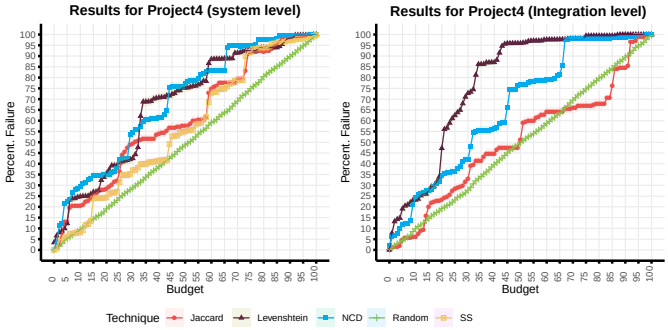

Fig. 6. Percentages of Failure found for Projects 4 on integration- and system-level. Budget values represent percentage of prioritized tests executed.

C. How long does it take to execute each technique on different level of testing?

We investigated RQ3 by first collecting the values of 10 execution times per technique on the system level, and once per project version on the unit level (See section III-A). We analysed Central Tendency and Variability of time in Table X, along with a post hoc statistical analysis in Table VII similar to the one described in RQ2.

1) Unit-level Execution Time: Levenshtein had the highest average execution time, followed by NCD, Jaccard, and Random. Random was faster than Jaccard by 48 sec, Jaccard faster than NCD by 2.5 min, and NCD faster than Levenshtein by 59.7% (4.5 min). All differences were reported as

TABLE VI
SUMMARY OF THE POST HOC STATISTICAL ANALYSIS ON DETECTED FAILURES ON SYSTEM LEVEL.

comp.	p value	Adj.p val	Kendall's W	Eff. Size	SSD
30% Budget					
Rand-SS	1.74E-13	1.74E-12	0.0059	S	Yes
Rand-Lev	<0.001	<0.001	0.0625	S	Yes
Rand-Jacc	<0.001	<0.001	0.1900	S	Yes
Rand-NCD	<0.001	<0.001	0.2373	S	Yes
SS-Lev	0.1776	>0.999	0.000013	S	No
SS-Jacc	1.44E-07	1.44E-06	0.0031	S	No
SS-NCD	<0.001	<0.001	0.2280	S	Yes
Lev-Jacc	9.16E-05	0.0009	0.0139	S	Yes
Lev-NCD	<0.001	<0.001	0.1635	S	Yes
Jacc-NCD	8.88E-16	8.88E-15	0.0603	S	Yes
50% Budget					
Rand-SS	<0.001	<0.001	0.1239	S	Yes
Rand-Lev	<0.001	<0.001	0.2638	S	Yes
Rand-Jacc	<0.001	<0.001	0.0108	S	Yes
Rand-NCD	<0.001	<0.001	0.3392	M	Yes
SS-Lev	<0.001	<0.001	0.2874	S	Yes
SS-Jacc	0.8347	>0.999	0.0120	S	No
SS-NCD	<0.001	<0.001	0.3243	M	Yes
Lev-Jacc	<0.001	<0.001	0.0584	S	Yes
Lev-NCD	0.0108	0.1081	0.0275	S	Yes
Jacc-NCD	<0.001	<0.001	0.0928	S	Yes
80% Budget					
Rand-SS	<0.001	<0.001	0.6155	L	Yes
Rand-Jacc	<0.001	<0.001	0.6298	L	Yes
Rand-Lev	<0.001	<0.001	0.7534	L	Yes
Rand-NCD	<0.001	<0.001	0.8125	L	Yes
SS-Jacc	0.2203	>0.999	0.0107	S	No
SS-Lev	1.67E-05	0.0002	0.0396	S	No
SS-NCD	3.66E-08	3.66E-07	0.0603	S	No
Lev-Jacc	0.0021	0.0207	0.0211	S	Yes
Lev-NCD	0.2297	>0.999	0.0013	S	No
Jacc-NCD	1.86E-05	0.0002	0.0843	S	Yes

largely significant by the post hoc test. The general trend for the standard deviation (SD) to increase as the average time increases, suggesting a correlation between average time and SD, possibly implying that techniques with a higher execution time are less reliable. Note, however, that some projects used had a large disparity in test suite size between versions, with project Math containing versions ranging from 820 to 4378 test cases, while some others had a low range. This could explain the varying SD.

2) Integration-level Execution Time: On integration level, Levenshtein took the most average time and with a high standard deviation that indicates a wide spread of the values. However, Jaccard' execution time was just ~49% (26 sec) slower than Random's. On the other hand, NCD had a mean time ~84% (1.5 hours) shorter than Levenshtein and ~95% (18 min) longer than Jaccard.

Furthermore, the post hoc statistical analysis states that these differences are significant with large/moderate effect size, besides NCD with Jaccard and Levenshtein.

3) System-level Execution Time: At a glance, the results on the system level follow a similar path to the unit level and a similar trend of the standard deviation such that techniques with higher execution times had higher SDs as well (except SS). Jaccard was ~87% (~10 min) faster than SS, SS ~55% (~15 min) faster than NCD, and NCD ~80% (~2 hours) faster than Levenshtein.

TABLE VII
SUMMARY OF THE POST HOC ANALYSIS ON THE EXECUTION TIME FOR ALL THREE LEVELS OF TESTING.

comp.	p value	Adj.p val	Kendall's W	Eff. Size	SSD
Unit-level					
Rand-Jacc	2.59E-10	1.55E-09	>0.999	L	Yes
Rand-NCD	<0.001	<0.001	>0.999	L	Yes
Rand-Lev	<0.001	<0.001	>0.999	L	Yes
Jacc-NCD	2.39E-08	1.44E-07	0.9329	L	Yes
Jacc-Lev	<0.001	<0.001	>0.999	L	Yes
NCD-Lev	2.59E-10	1.55E-09	>0.999	L	Yes
Integration-level					
Rand-Jacc	0.0833	0.4996	0.8837	L	No
Rand-NCD	0.0005	0.0032	0.9984	L	Yes
Rand-Lev	2.04E-07	1.22E-06	0.8851	L	Yes
Jacc-NCD	0.0833	0.4996	0.8438	L	No
Jacc-Lev	0.0005	0.0032	0.8844	L	Yes
NCD-Lev	0.0833	0.4996	0.9985	L	No
System-level					
Rand-Jacc	0.0796	0.7962	0.4096	M	No
Rand-SS	5.55E-10	5.55E-09	>0.999	L	Yes
Rand-NCD	7.24E-14	7.24E-13	>0.999	L	Yes
Rand-Lev	4.44E-16	4.44E-15	0.3492	M	Yes
Jacc-SS	8.60E-06	8.60E-05	0.9216	L	Yes
Jacc-NCD	1.00E-08	1.00E-07	>0.999	L	Yes
Jacc-Lev	1.50E-10	1.50E-09	0.4050	M	Yes
SS-NCD	0.2002	>0.999	0.2304	M	No
SS-Lev	0.0506	0.5056	0.2975	M	No
NCD-Lev	0.5002	>0.999	0.2975	M	No

The post hoc statistical tests reflected the differences between the techniques found. Random had no SSD with Jaccard, NCD had no SSD with Levenshtein and SS didn't have a SSD with neither NCD nor Levenshtein, despite being more rapid than both.

> RQ3: A clear trend is seen where the SD increases as the avg. time increases, except for SS. Generally, Jaccard is the fastest to execute and Levenshtein is the slowest.

D. How do different levels of testing affect the prioritization of a test suite?

Due to limitations in artefact availability, we could only compare integration and system levels. In addition, since SS was not as meaningful to execute on code, we exclude SS from the comparison across integration and system levels. The collected coverage data from RQ1 is used to interpreted from a different perspective. The failure and time data from RQ2 and RQ3 are then used in a Wilcoxon signed rank test to compare failure and time between integration and system level tests. The non-parametric paired test was used to identify if an SSD existed and measure the effectiveness given a Z–score z [28], [29] and the total number of datapoints N using the formula $\textit{EffectSize}(z, N) = \frac{z}{N^2}$.

Based on Project4's coverage in Fig. 4, we see similar curves on both levels, where Random records a higher coverage given a certain budget across all techniques (beside SS), followed by NCD, then Jaccard and Levenshtein with the lowest coverage.

In terms of detected failures, Fig. 6 reveals differences in techniques' effectiveness. This contrast between techniques are similar on both LoTs with Levenshtein having the highest rate and Random the lowest. However, the distinctions are clearer on integration level as the lines that represent the techniques are further apart. As such, Levenshtein's rate becomes higher and Jaccard's becomes worse. To verify the observations, a Wilcoxon signed rank test was performed on failure percentages over 115 builds and supported the results by showing an SSD between the two LoTs with a small effect size.

Finally, to complement the trade-offs of both LoTs, we describe the differences of techniques' execution times present in Table X which shows that techniques are generally faster to execute on system-level than integration level.

Through a Wilcoxon signed rank test, the difference between both levels is confirmed and reported to be significant with a large effect size as shown in Tables VIII and IX.

TABLE VIII
WILCOXON SIGNED RANKS TEST WITH TWO RELATED SAMPLES (INTEGRATION- AND SYSTEM-LEVEL FAILURE DETECTION RATE).

p value	Test statistics (W)	[Neg, Pos] Ranks	Z/N^2	Effect Size	SSD
0.000005	-372708	[1648994 , -2021702]	-0.0458	S	Yes

TABLE IX
WILCOXON SIGNED RANKS TEST WITH TWO RELATED SAMPLES (INTEGRATION- AND SYSTEM-LEVEL EXECUTION TIMES).

p value	Test statistics (W)	[Neg, Pos] Ranks	Z/N^2	Effect Size	SSD
6.58E-07	-740	[-780, 40]	-0.55603	L	Yes

RQ4: Coverage is similar on both LoTs. However Levenshtein's Failure detection rate is higher on integration level and Jaccard's is higher on system level. Also, all techniques but Levenshtein are faster on system level.

V. DISCUSSION

In this section, we discuss the results of each metric to have a better image of the techniques' attributes and behaviour on each of the three LoTs. Then we will interpret and further discuss the results of the comparison between integration- and system-levels of testing. Finally, we will provide some recommendations related to the usage of each technique based on the trade-off each technique presents.

TABLE X
TECHNIQUES' AVERAGE EXECUTION TIME (MINUTES), MEDIAN, AND STANDARD DEVIATION(SD) ON ALL THREE LEVELS, ROUNDED UP

	Jaccard	Lev	NCD	SS	Random
Cli					
Avg.	0.324	**0.875**	0.562	—	0.004
Median	0.3190	0.673	0.472	—	0.004
SD	0.027	0.513	0.255	—	0.002
Codec					
Avg.	0.763	**4.214**	1.188	—	0.006
Median	0.713	3.391	0.903	—	0.005
SD	0.087	2.093	0.563	—	0.002
Gson					
Avg.	0.49	**3.603**	3.194	—	0.012
Median	0.489	3.633	3.236	—	0.012
SD	0.024	0.453	0.408	—	0.0008
JacksonCore					
Avg.	0.413	**2.205**	0.865	—	0.005
Median	0.404	2.657	0.793	—	0.005
SD	0.0395	0.75	0.345	—	0.0014
JxPath					
Avg.	0.396	**1.651**	0.753	—	0.0048
Median	0.397	1.713	0.770	—	0.005
SD	0.008	0.257	0.064	—	0.0003
Lang					
Avg.	1.033	**33.575**	11.996	—	0.022
Median	0.931	30.7346	10.603	—	0.021
SD	0.23	10.016	3.737	—	0.003
Math					
Avg.	2.231	—	**30.485**	—	0.03
Median	1.979	—	25.5242	—	0.028
SD	1.287	—	20.619	—	0.011
Project 4 — Integration Level					
Avg.	0.906	**118.7004**	19.342	—	0.464
Median	0.88	103.923	19.106	—	0.455
SD	0.09	45.05	0.753	—	0.079
Project 1					
Avg.	1.473	**31.921**	15.615	13.364	0.26
Median	1.556	31.962	16.748	13.483	0.268
SD	0.328	0.648	3.491	3.455	0.051
Project 2					
Avg.	0.507	**36.640**	14.043	7.653	0.169
Median	0.51	37.527	14.294	7.712	0.155
SD	0.041	3.687	4.537	1.015	0.033
Project 3					
Avg.	3.556	**481.948**	64.676	20.531	0.627
Median	3.412	473.631	63.803	19.619	0.618
SD	0.292	22.775	1.924	7.07	0.049
Project 4					
Avg.	0.768	**31.898**	14.29	7.852	0.204
Median	0.681	31.134	12.594	7.879	0.200
SD	0.195	3.491	3.609	1.988	0.031

A. Coverage

To begin with, SS outperformed all techniques on a system level and showed a high requirement coverage on all relevant experimental subjects. This suggests that SS –specifically Doc2Vec-based approaches– are just as effective in test prioritization, in comparison to other fields and areas within software engineering where such SS approaches have been explored by researchers [21], [30], [31].

Moreover, we found that the coverage of a prioritized test suite was influenced by the distribution of the features. Projects 1-3 that were provided by CompanyA had respectively 31%, 8% and 33% of the total test cases linked to at least one requirement. Thus, the coverage graphs for projects provided by company A shown in Fig. 4 cover only a small portion of requirements, assuming that the rest of the tests (that miss a linked requirement) do not cover any requirement. Hence, the techniques in the Projects1-3 take a linear shape.

In contrast, Project4's test cases—provided by CompanyB—were all linked to exactly one requirement. In other words, coverage information was available for all the test cases. Therefore, a curve was clear in Project4's graphs in Fig. 4 on integration- and system-levels.

However, Random surprisingly outperformed Jaccard, Levenshtein and NCD. We traced the cause of this performance to some possible factors. Firstly, the Maximum mean algorithm used by the MultiDistances Package makes some decisions randomly on which string to prioritize when two strings are extremely similar. Consequently, the resulting ranking could be unrepresentative of the techniques. Ideally, each technique should be executed many times to control for randomness. However, we execute each technique (except Random) exactly once on each LoT.

Secondly, Project4 tests are not close in content to the requirements since the textual requirements are very short (2-4 words), and the tests are much longer (5-12 lines). This implies that finding the requirements words inside the test cases is harder, and hence, diverse tests, may not necessarily translate to diverse requirements coverage on both system and integration LoTs.

Finally yet importantly, we believe that due to the independent (non-hierarchical) treatment of requirements and sub-requirements, most of the them end up being covered by few test cases as shown by the histogram in Fig. 7. Therefore, DBT would consider a requirement and its sub-requirements as "not diverse" and avoid selecting both, whereas Random (x100) can randomly prioritize any requirement and its sub-requirements, hence covering more of them.

In short, the coverage data was skewed towards specific tests hindering conclusive results for diversity-based approaches. Moreover, the general curves produced by Project4 on both LoTs indicate that coverage is not influenced by altering the LoT rather than the prioritization techniques.

B. Failure-Detection Rate

We found that DBT were more effective than Random across all levels, with the exception of Jaccard on the integration level. This is in line with Henard et al.'s findings regarding the acceptable amount of faults even with restricted resources [7]. However, although DBT has an increasing effectiveness over Random, a closer look at the failure detection rate on the unit level (Fig. 5) reveals that there is not such a large difference between the DBT themselves. Furthermore, while all projects had a linear curve for Random, DBT had two distinct curves (See Section IV-B).

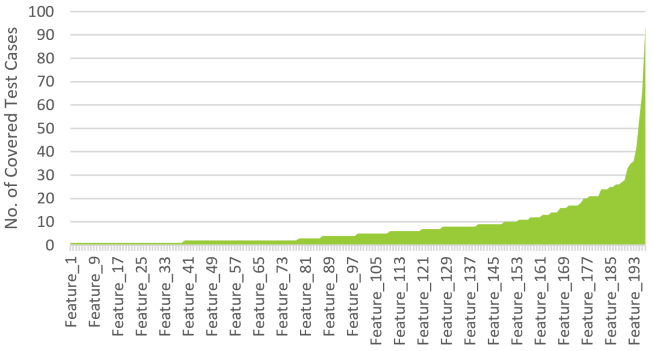

Fig. 7. Features/ Requirements distribution among 1605 test cases in Project4. Note that the features are just representations of 196 features in total.

Thus, this suggests that there are more complex properties at play in which some DBT are more sensitive than others, such as the low amount of textual information in each unit and system test method. For example, studies have indicated that NCD is not as effective when there is little information available [1], [5].

Diversity-based techniques' effectiveness vary depending on the content of the test, and the amount of inherited unit tests a project influence the performance described above since similarities between the tests (e.g., inherited members) are not found in the test file. Projects Cli and Math (Fig. 5), for example, had a number of triggering tests that were inherited in several versions, which could contribute to obtaining a more parabolic curve. Executing one inherited test essentially runs the rest of the identical tests as well, possibly explaining the distinct curve all three DBT have at Lang from the 0-5% budget mark (Fig. 5).

While the integration and system level failure information are difficult to compare equally due to the scarcity of projects, the individual DBT techniques have some visual differences. On top of that, according to the histogram in Fig. 8, failure detection rate and requirement/feature coverage are correlated in Project4 on integration and system-levels. Given that each test case covers exactly one requirement, the histogram shows that a few certain requirements trigger up to 550 failures each, indicating skewed failure data. Note that test suites with higher requirement coverage does not necessarily mean they cover the triggering tests (since they would have to cover specific features). Consequently, the technique that performs best with regard to coverage, does not necessarily have high detection rate and vice versa. Moreover, Hemmati et al.'s results concluded that the best case scenario for Diversity-

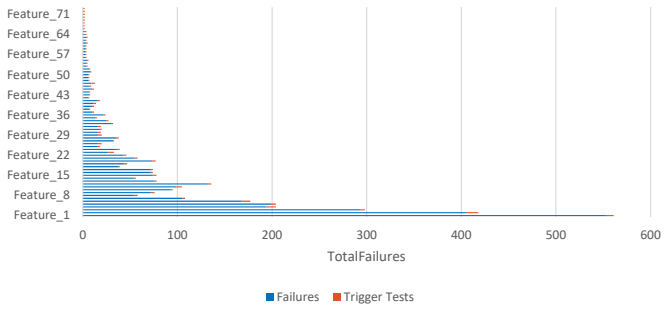

Fig. 8. The total number of failures and trigger tests related to each feature in Project4 over 669 builds. Most of the triggering tests are connected to Feature 1. Note that the features are just representations of 196 features in total

based test selection in terms of effectiveness is when the triggers that detect discrete faults are diverse [19]. As such, DBT can obtain both a high coverage and failure detection rate if different failures are triggered by tests that cover diverse requirements.

C. Execution Time

According to the results in Tables X and with respect to the projects' sizes in Table III, we can see that although two projects are of similar sizes, DBT take longer time to execute on one than the other. For instance, Project1 and 3 have respectively 2639 and 2691 test cases. However, DBT are much slower in general on project3. In fact, Levenshtein is x15 slower on project3 than project1. Another example is Gson that have more than twice as many test cases as Codec but has even a slightly lower execution time. The reason is most likely the length of the individual test cases. Regardless the number of test cases, the number of words or line of codes (LoC) influence the number of operations performed on the test and thus the speed of the technique. A similar pattern is seen in the results of Henard et al.'s study, where SUTs with a smaller number of test cases, but with more lines of code take longer to execute on black box techniques in general [7].

D. Recommendations

Below, we condense the outcomes of this experiment into bulleted recommendations for users of these techniques. We also present some recommendations in Table XI regarding the application of specific diversity measures for different LoT.

- If the number of test cases to prioritize are below 20% or over 90% of the test suite, Random is recommended as it doesn't differ in terms of effectiveness from other techniques on these budgets, yet, it's significantly faster.
- If requirements coverage is the priority of the practitioner, the distribution of the requirements among test cases should be checked before executing test prioritization. If the coverage is skewed towards specific features (as in Fig. 7), then we recommend an approach which executes a cheaper technique such as Jaccard on tests covering the subset of features that have similarly distributed coverage. For instance, from Feature_1 to Feature_161 in Fig. 7. Another effective technique on the current LoT can then be executed on the rest of the tests, e.g. Feature_169 to Feature_193. A few prioritized tests can be executed from the skewed features, and the majority of prioritized tests can be executed from the normally distributed features. This would reduce the total execution time and increase coverage, especially for DBT which scale expensively. However, if splitting the prioritization is not possible, then we strongly recommend SS, as it provides the highest coverage over all system-level projects.
- If failure detection rate is the priority of the practitioner, a history of previous failures is suggested to be studied beforehand to understand the nature of the failures distribution among test cases or requirements as in Fig. 8. This can be used as a guideline to choose the budget of the test suite to prioritize. Similar recommendations are used when performing history-based approaches [32].
- Before deciding the prioritization technique, check the number of total test cases as well as the size of the test suite in bytes. Although projects may have the same number of tests, the content of each test can be larger, thus the size of the test suite is larger and may significantly increase the execution time. On all levels, it is generally not recommended to use Levenshtein due to its expensive execution time and high unreliability.
- We suggest cleaning the system-level tests from unsound data, such as invalid strings, for a better performance of the prioritization techniques based on textual analysis.

VI. Validity Threats

In a fractional factorial experiment, several inherent limitations exist. As SS was not executed on integration level, it was not possible to include SS in the comparison between integration and system level. The power of statistical tests may be relatively weak as well, since some samples on the unit level were quite small compared to industrial software.

The companies that we studied originated in a close geographical area. As companies have internal policies on the creation and maintenance of artifacts, these findings are context specific, further lowering our external validity. However, the two companies that we studied had distinct industrial focuses, and have expanded globally to become international companies, reducing the effect of this weakness.

Our experiment was impacted by several aspects. First, by our execution, which had to be altered due to time and resource limitations. Time constraints further prevented us from executing Levenshtein on projects with large test suites, namely Math. This issue was caused by the project size compounded by number of versions. Since each version had a unique test suite, executing only a section of the versions would skew the average if only half, for example, of the versions were executed. However, there is still a significant amount of projects with Levenshtein executed on unit level.

Time constraints limited the control of randomness too when using Maximum Mean algorithm provided by the MultiDistances package. Therefore, each technique should ideally be

TABLE XI
RECOMMENDATIONS OF LOTS TO USE GIVEN AN A-DIV TECHNIQUE AND ITS TRADE-OFF.

Technique	Trade-off	Recommended LoT
Jaccard	Fast to execute on all LoTs. Bad coverage on system level. Good Failure detection on unit level but bad on system and integration levels.	Unit level
Levenshtein	Fast on unit level. Bad coverage on system level. Best Failure detection on integration level, worst Failure detection on unit level	Integration level
NCD	Slowest on Integration level. Bad coverage on system level. Good Failure detection on all LoTs.	Unit/ System level
SS	Relatively Fast. Good coverage, Moderate Failure detection rate. Can only be applied on system level.	System level
Random	Fast to execute on all LoTs. Good coverage. Bad failure detection.	When $\leqslant 30\%$ or $\geqslant 90\%$ budget

executed an equal number of times on every level of testing. Yet, for failure detection, while Random was executed 100 times per project version, the other techniques were only executed once per version on the unit level and 10 times on the integration and system level.

Furthermore, the number of versions for each project ranged from 18 to 106, decreasing internal validity. This was mitigated by using projects of differing sizes and versions, increasing generalizability. Ultimately, Jaccard and NCD were executed 293 times, and Levenshtein 123 times across all projects on the unit level, which is still a substantial amount. Similarly, for coverage, Random was executed 100 times but only once for other techniques on the system and integration level.

These constraints also required us to use different machines on Unit, Integration and System level. Since each host machine could have different background applications running, the execution time could have been unstable. This was mitigated by sequentially running the techniques on a Virtual Machine.

Due to the unavailability of suitable projects on integration LoT, only one project was used, reducing the representativeness and reliability of the results on the integration level, leading to a high variability. Nevertheless, the project used was an active project from an industry partner, increasing the external validity of this particular level of testing.

Being an experiment first and foremost, this study may have a relatively low external validity, with Defects4J providing a controlled environment on the unit level. However, several decisions were made to the experimental design to mitigate this. Active projects from industry partners were used on both the system and integration LoT, providing realistic test objects.

Our choice of instrumentation could also impact our internal validity. MultiDistances' technique implementation could possibly be faulty, providing incorrect rankings. However, we, along with our supervisor, have reviewed the implementation and have not found obvious faults.

VII. CONCLUSION

This paper set out to compare 4 Diversity-based prioritization techniques (DBT) namely Jaccard, Levenshtein, NCD, and Semantic Similarity on three levels of testing (i.e. unit-, integration-, and system-level) in terms of coverage, failure rate detection and execution time. Through an experiment, we have found that some techniques perform better on a specific level of testing given a certain budget. We have also uncovered possible test suite properties that affect diversity based techniques, such as the spread of tests on system features that affect the execution of a-div techniques. While we have specifically focused on prioritization using Artefact-based diversity (a-div) techniques as DBT, the relation between an uneven spread of tests on features and a-div techniques implies that our findings are likely to be of importance to other diversity based techniques such as behavioural-based diversity. In terms of future research, we particularly suggest implementing Semantic Similarity on unit and integration levels, comparing a wider range of diversity-based techniques including behaviour-based against a-div techniques, and using more test subjects on the integration level. Furthermore, the effects of system test structure, such as the inclusion of unsound strings, on a-div techniques can be studied in more detail. Investigating the effects of the number of tests versus the size of tests on different LoT would be interesting as well.

VIII. ACKNOWLEDGEMENT

The authors would like to acknowledge their families, the existence of memes, and the obscene amounts of coffee which have provided support and motivation during hard times. The authors would also like to express their special thanks to Francisco Gomes Oliveira Neto and Gregory Gay for providing guidance, clarifications, and assistance with this paper.

REFERENCES

[1] F. G. de Oliveira Neto, A. Ahmad, O. Leifler, K. Sandahl, and E. Enoiu, "Improving continuous integration with similarity-based test case selection," in *Proceedings of the 13th International Workshop on Automation of Software Test*, pp. 39–45, 2018.

[2] F. G. de Oliveira Neto, R. Feldt, L. Erlenhov, and J. B. D. S. Nunes, "Visualizing test diversity to support test optimisation," in *2018 25th Asia-Pacific Software Engineering Conference (APSEC)*, pp. 149–158, IEEE, 2018.

[3] R. Feldt, S. Poulding, D. Clark, and S. Yoo, "Test set diameter: Quantifying the diversity of sets of test cases," in *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, pp. 223–233, IEEE, 2016.

[4] Y. Ledru, A. Petrenko, S. Boroday, and N. Mandran, "Prioritizing test cases with string distances," *Automated Software Engineering*, vol. 19, no. 1, pp. 65–95, 2012.

[5] 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*, pp. 178–186, IEEE, 2008.

[6] F. G. de Oliveira Neto, F. Dobslaw, and R. Feldt, "Using mutation testing to measure behavioural test diversity," in *the 15th International Workshop on Mutation Analysis. To appear*, 2020.

[7] C. Henard, M. Papadakis, M. Harman, Y. Jia, and Y. Le Traon, "Comparing white-box and black-box test prioritization," in *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, pp. 523–534, IEEE, 2016.

[8] M. Bilenko and R. J. Mooney, "Adaptive duplicate detection using learnable string similarity measures," in *Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining*, pp. 39–48, 2003.

[9] Q. Le and T. Mikolov, "Distributed representations of sentences and documents," in *International conference on machine learning*, pp. 1188–1196, 2014.

[10] H. Hemmati, Z. Fang, and M. V. Mantyla, "Prioritizing manual test cases in traditional and rapid release environments," in *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*, pp. 1–10, IEEE, 2015.

[11] E. G. Cartaxo, P. D. Machado, and F. G. O. Neto, "On the use of a similarity function for test case selection in the context of model-based testing," *Software Testing, Verification and Reliability*, vol. 21, no. 2, pp. 75–100, 2011.

[12] A. Arcuri and L. Briand, "Adaptive random testing: An illusion of effectiveness?," in *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, pp. 265–275, 2011.

[13] B. Miranda, E. Cruciani, R. Verdecchia, and A. Bertolino, "Fast approaches to scalable similarity-based test case prioritization," in *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, pp. 222–232, IEEE, 2018.

[14] P. Jaccard, "Étude comparative de la distribution florale dans une portion des alpes et des jura," *Bull Soc Vaudoise Sci Nat*, vol. 37, pp. 547–579, 1901.

[15] G. Casanova, E. Englmeier, M. E. Houle, P. Kröger, M. Nett, E. Schubert, and A. Zimek, "Dimensional testing for reverse k-nearest neighbor search," *Proceedings of the VLDB Endowment*, vol. 10, no. 7, pp. 769–780, 2017.

[16] J. H. Lau and T. Baldwin, "An empirical evaluation of doc2vec with practical insights into document embedding generation," *arXiv preprint arXiv:1607.05368*, 2016.

[17] Q. Chen and M. Sokolova, "Word2vec and doc2vec in unsupervised sentiment analysis of clinical discharge summaries," *arXiv preprint arXiv:1805.00352*, 2018.

[18] S. Yoo and M. Harman, "Regression testing minimization, selection and prioritization: a survey," *Software testing, verification and reliability*, vol. 22, no. 2, pp. 67–120, 2012.

[19] H. Hemmati, A. Arcuri, and L. Briand, "Empirical investigation of the effects of test suite properties on similarity-based test case selection," in *2011 Fourth IEEE International Conference on Software Testing, Verification and Validation*, pp. 327–336, IEEE, 2011.

[20] H. Hemmati, A. Arcuri, and L. Briand, "Reducing the cost of model-based testing through test case diversity," in *IFIP International Conference on Testing Software and Systems*, pp. 63–78, Springer, 2010.

[21] S. Tahvili, M. Ahlberg, E. Fornander, W. Afzal, M. Saadatmand, M. Bohlin, and M. Sarabi, "Functional dependency detection for integration test cases," in *2018 IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C)*, pp. 207–214, IEEE, 2018.

[22] I. Markov, H. Gómez-Adorno, J.-P. Posadas-Durán, G. Sidorov, and A. Gelbukh, "Author profiling with doc2vec neural network-based document embeddings," in *Mexican International Conference on Artificial Intelligence*, pp. 117–131, Springer, 2016.

[23] 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 2014 International Symposium on Software Testing and Analysis*, pp. 437–440, 2014.

[24] S. Elbaum, A. G. 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.

[25] M. Kuhrmann, K. Schneider, D. Pfahl, S. Amasaki, M. Ciolkowski, R. Hebig, P. Tell, J. Klünder, and S. Küpper, "Product-focused software process improvement," 2018.

[26] M. Staats, M. W. Whalen, M. P. Heindahl, and A. Rajan, "Coverage metrics for requirements-based testing: Evaluation of effectiveness," 2010.

[27] F. G. de Oliveira Neto, R. Torkar, R. Feldt, L. Gren, C. A. Furia, and Z. Huang, "Evolution of statistical analysis in empirical software engineering research: Current state and steps forward," *Journal of Systems and Software*, vol. 156, pp. 246–267, 2019.

[28] A. Field, *Discovering statistics using IBM SPSS statistics*. sage, 2013.

[29] A. Aron and E. N. Aron, *Statistics for psychology*. Prentice-Hall, Inc, 1999.

[30] S. Tahvili, L. Hatvani, M. Felderer, W. Afzal, and M. Bohlin, "Automated functional dependency detection between test cases using doc2vec and clustering," in *2019 IEEE International Conference On Artificial Intelligence Testing (AITest)*, pp. 19–26, IEEE, 2019.

[31] B. M. S. Misra, A. M. A. R. C. Torre, J. G. R. M. I. Falcão, D. T. B. O. Apduhan, and O. Gervasi, *Computational Science and Its Applications–ICCSA 2019*. Springer, 2019.

[32] A. Haghighatkhah, M. Mäntylä, M. Oivo, and P. Kuvaja, "Test prioritization in continuous integration environments," *Journal of Systems and Software*, vol. 146, pp. 80 – 98, 2018.