Evaluating the Trade-offs of Text-based Diversity in Test Prioritisation

Ranim Khojah, Chi Hong Chao, Francisco Gomes de Oliveira Neto
Department of Computer Science and Engineering
Chalmers | University of Gothenburg
Gothenburg, Sweden
khojah@chalmers.se, guschaoch@student.gu.se, francisco.gomes@cse.gu.se

Abstract—**Diversity-based techniques (DBT) have been cost-effective by prioritizing the most dissimilar test cases to detect faults at earlier stages of test execution. Diversity is measured on test specifications to convey how different test cases are from one another. However, there is little research on the trade-off of diversity measures based on different types of text-based specification (lexicographical or semantics). Particularly because the text content in test scripts vary widely from unit (e.g., code) to system-level (e.g., natural language). This paper compares and evaluates the cost-effectiveness in coverage and failures of different text-based diversity measures for different levels of tests. We perform an experiment on the test suites of 7 open source projects on the unit level, and 2 industry projects on the integration and system level. Our results show that test suites prioritised using semantic-based diversity measures causes a small improvement in requirements coverage, as opposed to lexical diversity that showed less coverage than random for system-level artefacts. In contrast, using lexical-based measures such as Jaccard or Levenshtein to prioritise code artefacts yield better failure coverage across all levels of tests. We summarise our findings into a list of recommendations for using semantic or lexical diversity on different levels of testing.**

Index Terms—**Diversity-based testing, Natural Language Processing (NLP), Test Case Prioritisation**

I. Introduction

Testing is crucial in a software-intensive system to ensure a satisfactory degree of quality. However, test resources are often limited, particularly due to the need for short and continuous feedback cycles in modern software development. Automated test prioritisation approaches aid testers to decide on what and how much to test in order to achieve cost-effective testing. Several types of test case prioritisation exist depending on prioritisation criteria. Specifically, Diversity-based Testing (DBT) can reduce test costs with a satisfactory test coverage [1], [2], and increase fault detection rate [3], [4]. Test case prioritisation involves determining the order of execution for all tests, whereas test case selection aims to identify subsets of tests. Diversity is measured through distance functions that convey how different two pieces of information are from one another.

Literature include many proposed strategies to measure diversity, each with their own contributions and limitations when it comes to applicability, performance and domain suitability. Diversity is also investigated in relation to other relevant aspects of software testing or maintenance such as identifying dependencies between requirements [5], [6], or visualising redundancies in test suites [3]. However, literature rarely explores the trade-off in applying these techniques at different levels of testing (unit, integration or system). Levels of testing are groupings of tests in different stages of the software development lifecycle. 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 diversity measure may drastically impact test prioritisation performance.

Typically, diversity is measured on textual artefacts such as requirements, test scripts (code or natural language), or system output to determine the diversity (i.e., dissimilarity) between tests [7], [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 [8]. This may result in inaccurate test suite prioritisation [9]. Therefore, we apply Natural Language Processing (NLP) to measure the *semantic diversity* between test specifications. Unlike lexical similarity which only considers actual characters, a semantic-based approach considers the context in which the word such as the order between words, or whether they are synonyms. For instance, the words ‘begin’ and ‘start’ are semantically similar but lexically dissimilar. Since the language used in writing test artefacts vary significantly between unit/integration (code) and system/acceptance (domain-specific or natural language), we aim to understand the trade-off of using semantic similarity to prioritise those different tests.

We perform an experiment to observe the trade-off of different text-based diversity on effectiveness (requirements coverage and failure) and efficiency when prioritising tests. We prioritise tests from 7 open source projects and 2 projects from two companies on three levels of testing. Particularly, we use Doc2Vec which is a neural network model that encodes documents [10] and has been showed promising results in semantic similarity tasks [11]. For lexical similarity, we use Jaccard and Levenshtein distances to prioritise tests [1], [12].

Our contributions are:[1]

- An experimental study that investigates the applicability, performance, and cost of several text-based diversity techniques, on three levels of testing. Our instrumented workflow is available in Github and can be adapted in future studies to compare results for coverage, failure detection rates and techniques' execution times for Python and Java projects.
- An implementation of our NLP-based prioritization techniques which makes use of Doc2Vec [10] and the Cosine Distance to rank a test suite. These techniques can be used by practitioners to prioritise their own test suites.
- A list of recommendations to use certain text-based techniques for test artefacts at different levels of testing.

II. Text-based Diversity on Test Artefacts

Next, we illustrate how lexical or semantic diversity affect test prioritisation at different levels of testing. Our system under test (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 6 test cases. Note that testEggNum() and testIsEggEmpty() exercise similar quality scenarios to each other. Likewise for the pair testMilkNum() and testIsMilkEmpty(). This is perhaps shown more clearly in the system level tests, which are written in a 'Given-When-Then' syntax supported by tools such as Cucumber[2]. On the system level, one can easily *read* which scenarios are related to eggs [”Number of Eggs”, ”Egg Status”] or milk [”Number of Milk”, ”Milk Left in Farm”].

Given that there are limited resources to execute, e.g., 3 out of 6 tests on each level, we aim to cover all features. While there is no one right answer, a valid answer could be to run [getChickens(), getMilkNum(), isEggEmpty()] on the unit level because they cover distinct attributes of the class. It should be noted that tests of different levels of testing 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 lexical (tokens) diversity on the test methods' names can automatically identify which tests to run to cover different scenarios, but reality is often more complex. For instance, lexical alone can lead to a suboptimal prioritisation of having both [”Number of Eggs”, ”Egg Status”] system level tests being chosen first, instead of a combination between Eggs and Milk. Factors such as different test authors, or synonyms in different tests can make the text-based technique less diverse tests. For instance, the naming of tests may follow different guidelines (e.g., start with testX, or shouldX), or use domain-specific terms). Therefore, context-awareness might enhance prioritisation. NLP is capable of revealing semantic differences and determine, e.g., that ”Egg Status” and ”Number of Milk” should be prioritized first.

On the other hand, there might be a point of diminishing returns in running costly NLP 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 lexical approach. This is especially true in the Farm example,

[1]The experiment workflow and implementations of techniques described below are available in our Github repository: https://github.com/ranimkhojah/test-prioritization-tools

[2]https://cucumber.io/docs/gherkin/

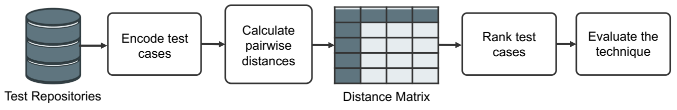

Fig. 1. Our instrumentation to perform diversity-based test prioritisation.

as the features are similar in implementation (isEggEmpty() and isMilkEmpty() are nearly identical).

A. Diversity-based Prioritisation

Diversity-based test case prioritisation has contributed to automated test optimization by enhancing the coverage at a low cost [1], and supporting data-driven decision making on test maintenance [3]. Studies also showed 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 [4], [13]–[16].

The diversity is determined by a distance function that calculates how dissimilar two pieces of information are. In our experiment, we use distance functions that capture similarities by following the process in Fig. 1. Starting with a test suite, 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 is 0, and totally dissimilar if their distance is 1. These pairwise distance values are arranged in a matrix, which can be used for other testing activities.

There are numerous techniques that measure the diversity of text information, ranging from simple token matching [3], [8], to positional encoding by preserving the order and context of words [5], [6], and, lastly, transformer models applied to large language models (e.g., GPT or BERT) that have self-attention [17]. Note that our goal is not to generate natural language test cases, hence transformers would be too costly for our goal. Rather, the test artefacts were already written by a tester, so we argue that approaches with positional encoding of the words are sufficient for our investigation. Alternatively, measures such as the Normalised Compressed Distance (NCD) can be applied to any information on the byte level and has been widely used to prioritise test cases in literature [1], [4]. Particularly, we focus on the following measures: Jaccard's Index, Levenshtein distance, NCD, and Semantic Similarity. Next, we detail each measure, whereas a summary of individual characteristics shown in Table I.

Jaccard's Index [18] 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 many n-grams the test cases share.

Levenshtein distance, or edit distance, between two strings A and B is the minimal number of operations (character insertion, deletion, replacement) to change string A into B. 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.

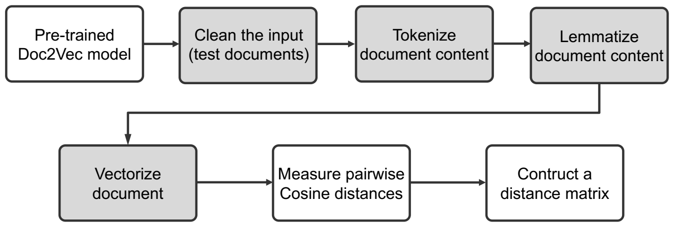
Pre-trained Doc2Vec model
Clean the input (test documents)
Tokenize document content
Lemmatize document content
Vectorize document
Measure pairwise Cosine distances
Contruct a distance matrix

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

Unlike Jaccard, the string is not ‘broken’ into tokens, hence the order in which characters appear when comparing pairs of words will affect the distance value.

The **Normalized Compression Distance (NCD)** [19] between two entire *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. Higher compression rates indicate more reptition between those two pieces of information, i.e., x and y have more similar content. Note also that the compression rate is also affected by the length of the documents being compared, such that longer documents (test cases) are more likely to include redundancy, independently of that redundancy being *between* the two documents.

We refer to **Semantic Similarity** as our technique that goes beyond lexical similarity between tests. We use the steps in a Natural Language Processing (NLP) technique (Fig. 2) to capture the distances between documents semantically. Since NLP assumes the presence of natural language, we only analyse Semantic Similarity for system-level artefacts as the unit and integration level used in our experiments carry code information.

The system-level test specifications used here describe the main purpose of a given test case along with its conditions, steps and the expected outcome. These test specifications go through the Semantic Similarity pipeline which performs the following processes: Cleaning which removes non-Latin characters, URLs, punctuation and stop words, e.g., pronouns or conjunctions; Tokenization that converts sentences to lists of words; Lemmatization that converts each token to its root.

Once the test specifications are processed, we capture semantic similarities between test cases using Doc2Vec (an unsupervised framework introduced by Le and Mikolov [10]). We use an off-the-shelf Doc2Vec model that is pre-trained on Wikipedia data[3], which covers the knowledge required to get our model to understand natural language. Doc2Vec uses the Paragraph Vector algorithm to construct a vector representation of each document, the vector captures features of a document with respect to the words' semantics and ordering. Then, Doc2Vec uses a built-in function to compute the Cosine distance [20], [21] that we use to measure the pairwise distances among all test cases in a test suite, and then arrange them in a distance matrix.

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

TABLE I
SUMMARY OF THE TEXT-BASED DIVERTSITY MEASURES USED IN OUR 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 cases and creates vectors for each document then uses 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.

III. Related Work

Many studies have looked into test case prioritization. Yoo and Harman [22] analyzed trends in regression test case prioritization showing its 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. [2] 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. Hemmati et al. [23], [24] looked 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.

Besides fault-revealing capabilities, test diversity has been a helpful criteria in revealing additional information from test repositories, such as visualization of test redundancy [3], dependencies between system requirements [5] and how to parallelise test execution [25].

There are numerous ways to extract diversity measures from different artefacts and using different distance functions. Feldt et al., [12] presented a model for a family of universal, practical test diversity metrics. Most diversity-based testing work still leverages on comparing strings. Aman et al. [26] employ a morphological analysis technique to preprocess test cases, which involves selecting only verbs, nouns, and adjectives from the test case document. They then utilize the Jaccard index to compute pairwise distances between test cases. Subsequently, they apply a clustering algorithm to group similar test cases, facilitating the selection of test cases for regression testing.

Although rare, literature include some studies capturing semantic similarities . Yang et al. [27] conducted a case study to evaluate the use NLP in 3 types of test case prioritization — including diversity-based. The results show that NLP outperform Random in all 3 types, but NLP still showed the worst performance in diversity-based test case prioritization in comparison with the other 2 types.Tahvili et al. [5] presented a NLP approach that revealed dependencies between requirements specification by analyzing the test case descriptions. 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. Doc2Vec has also been used to classify test cases into dependent or independent [6], [28], such that incorporating clustering can help testers to classify tests based on their functional dependencies. We also consider Doc2Vec, but we do not compare specific clustering technique as we rank tests based on their distance values from the distance matrix. Moreover, we compare text diversity on code and natural language.

In short, existing studies report that NLP provides benefits to support testers, but also limitations such as little improvement [27], or the need to be combined with clustering approaches [6]. Our paper adds to these findings by investigating text diversity on distinct types of text (code or natural language) in relation to different levels of testing.

IV. Methodology

We perform a fractional factorial experiment to evaluate the trade-offs of diversity measures on textual test artefacts. A fractional factorial experiment is a statistical experiment where only a subset of the factors are considered in a larger design space (i.e., the range of possible values of variables in the experiment). In this study, the design space includes diversity measures on textual test artifacts, which are evaluated by varying two factors: (i) distance measures, and (ii) levels of testing. The distance measures used in the experiment are: Jaccard, Levenshtein, NCD, Semantic Similarity, and a test suite in a randomised priority (Random) as baseline. We compare three levels of testing: Unit, integration and system.

TABLE II

SUMMARY OF THE SYSTEM UNDER TESTS (SUTS). SINCE WE ANALYSE UNIT-LEVEL PROJECTS ACROSS MANY VERSIONS, WE INDICATE THE MEAN NUMBER OF TEST CASES (TCS) PER VERSION.

SUT	Num. of TCs (Faults)	Source	Testing Level
Project 1	2691 TCs	CompanyA	System
Project 2	1605 TCs	CompanyB	System/Integration
Cli	262 TCs (39 faults)	OpenSource	Unit
Codec	440 TCs (18 faults)	OpenSource	Unit
Gson	988 TCs (18 faults)	OpenSource	Unit
JacksonCore	356 TCs (26 faults)	OpenSource	Unit
JxPath	347 TCs (22 faults)	OpenSource	Unit
Lang	1786 TCs (64 faults)	OpenSource	Unit
Math	2513 TCs (106 faults)	OpenSource	Unit

Unit-level test artefacts considered here are *test methods* written in a xUnit framework (e.g. JUnit) that test a class. We use integration tests scripts calling *API endpoints* to the SUT's modules. System-level tests are written in *natural language* and describe the user actions and expected systems output. Semantic Similarity is only applied to system-level artefacts since it the Doc2Vec model was not trained on programming languages. The other diversity-based techniques however, are used on all levels of testing. Our research questions are:

- **RQ1:** How do different text-based diversity measures perform in terms of requirements coverage?
- **RQ2:** To what extent does each diversity measure uncover failures?
- **RQ3:** How long does it take to execute each technique on different level of testing?

The experiment executes each technique on certain levels of testing following the process in Fig. 1. The response variables in our experiment are: coverage, execution time, and failure detection rate.

To instrument the prioritisation based on lexical diversity, we used the MultiDistances package[4] which has been used in other studies [3], [4], [9]. The package reads test cases as separate files in a folder, creates a distance matrix, and outputs a rank of the the test suite with regards to the chosen distance measure. MultiDistances does not include semantic diversity, so we instrumented an NLP-based approach using an off-the-shelf Doc2Vec model according to the process in Fig. 2.

We use open source data from Defects4J[5] [29] that provides unit tests that detect isolated faults along with specific information regarding tests that trigger such faults. Additionally, we use data from two industry partners, and open source projects in GitHub (Table II). 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. Due to non-disclosure agreements, we cannot include the test artefacts used in our experiments. A generic example is shown below that represents the integration- (code) and system-level (docstring) artefacts from both companies.

[4]https://github.com/robertfeldt/MultiDistances.jl

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

```python
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

We measure feature coverage by using the traceability information of each test on the integration and system levels and its corresponding requirement.[6] 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) [30], i.e., how early the prioritized test suite detect failures. Finally, we compare the time required to perform the prioritization–including the distance matrices generation–to help addressing a bigger picture of the trade-off in each technique. Although we need to adjust data collection to each level of testing, note that the same metrics are used for all levels of testing (with a few exceptions detailed below).

Unit-level Data: The Defects4J framework was selected due to its large collection of real, reproducible faults, each with documented properties and triggering tests. We use a total of seven open-source projects that differ in size as test subjects on the Defects4J framework. Defects4J isolates faults in different versions of the software, each containing a different test suite (as both the system and test suite evolved), hence we cannot merge all faults into a single version. Therefore, we: 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 levels of testing—which do not have fault information available—we count the total of failures, instead of faults.

Integration-level Data: Each of the 1605 integration tests could be traced to a single system-level test as well as a single requirement. Project 2 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.

System-level Data: The data gathered on a system-level was obtained from Projects 1 and 2. Since the test cases are written by human testers, there are many test case specifications that either do not follow a standard (e.g., have missing expected

[6]We use the term feature and system requirement interchangeably throughout the paper. Both companies use those two terms to address functionalities at the system-level.

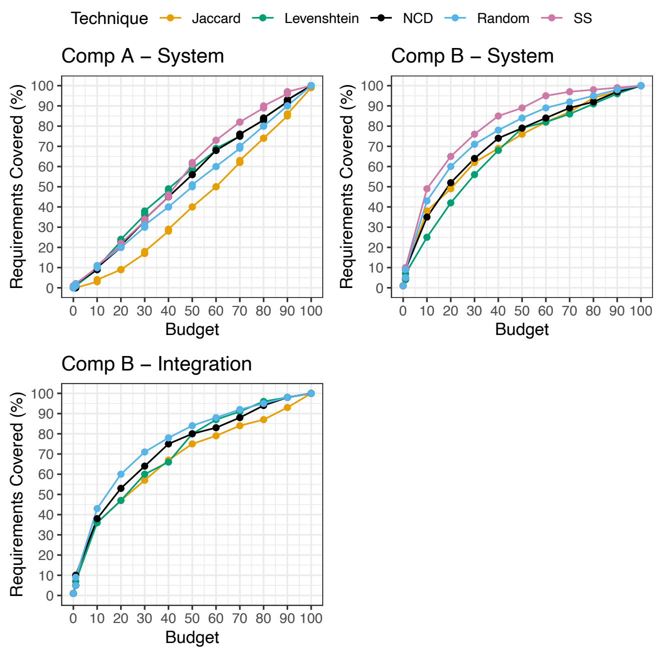

Fig. 3. The plots show the percentage of coverage for Projects 1 and 2. Budget values represent percentage of prioritized tests executed. There were 1281 requirements for Comp A, and 197 for Comp B. (SS = Semantic Similarity)

outputs, or incomplete actions) or are duplicates of other test case specifications. Company A did not provide information on whether their tests failed, therefore, failure detection rate was measured only for Project 2 at the system-level.

Execution of the experiment: 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 the system and integration level were executed **10 times** per project to account for variations in the prioritisation, except for the Randomised prioritisation which is executed 100 times.

V. Results and Analysis

A. *RQ1: Comparing Requirements Coverage*

Fig. 3 illustrates the percentage of covered features for a given number of test cases (Budget) using different text-based diversity techniques on integration and system level and a random prioritisation as a baseline. For system-level coverage for the project provided by Company A, all the techniques took a linear shape which indicates an overall low performance in feature coverage, but most techniques can cover slightly more requirements than Random when executing more than 20% of the tests (circa 5–10% more requirements). Company B shows a different shape and results where Semantic Similarity is the only technique that has higher coverage than Random. At the integration level, Random was slightly better than all lexical-based techniques.

A closer look at the test cases show that there are different distributions in the traceability between tests and requirements which affect the different shapes in both projects (discussed further in Section VI). However, Semantic Similarity was the only technique able to perform better than Random in both cases.

RQ1: Overall, none of the techniques had significantly better coverage than Random. Semantic diversity performed better by covering between 5–10% more requirements when more than 20% of the tests are executed.

B. *RQ2: Comparing Failure Detection*

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 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.

Failures on Unit-level: Across all projects (Fig. 4), 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. Our post hoc analysis (Table III) 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, 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 lexical techniques have a smaller effect size amongst themselves.

Failures on Integration-level: The failure detection rate of the techniques is similar for small test budgets, as shown in Fig. 5 (right). 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. For brevity, we moved the results of our statistical analysis on the integration level to our Github repository. However, the results align with the visual analysis where Levenshtein performs significantly better than NCD followed by Jaccard. The effect between them is larger when compared to Random, particularly after executing 50% of the test suite.

Failures on System-level: For system-level in Fig. 5 (left), Semantic Similarity 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.

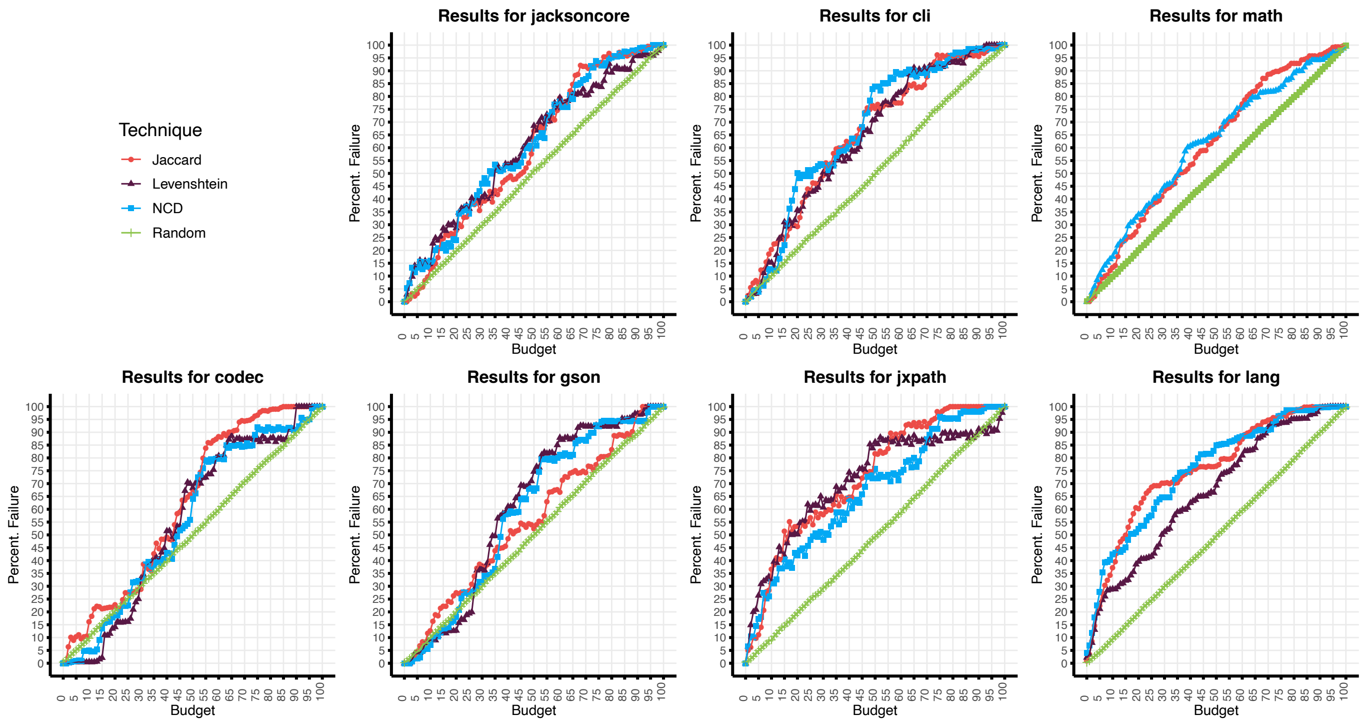

Fig. 4. Percentage of Failures found for all open source projects studied. Budget values represent percentage of prioritized tests executed.

TABLE III
SUMMARY OF THE POST HOC ANALYSIS ON DETECTED FAILURES ON UNIT LEVEL WHERE EACH ROW REPRESENTS A PAIRWISE COMPARISON. (SSD = STATISTICAL SIGNIFICANT DIFFERENCE)

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

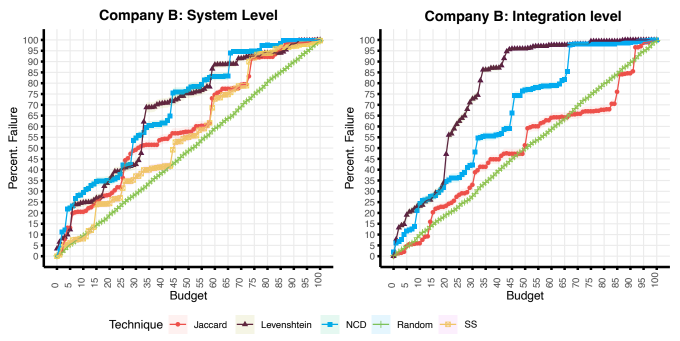

Fig. 5. Percentages of Failure found for Company B projects on integration- and system-level. Budget values represent percentage of prioritized tests executed. (SS = Semantic Similarity)

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

RQ2: No technique consistently finds more failures on all levels of testing. However, there is a greater distinction between the techniques and Random as budget increases up until 80%. Semantic Similarity has SSD with the lexical techniques, except with Jaccard.

TABLE IV
SUMMARY OF THE POST HOC STATISTICAL ANALYSIS ON DETECTED FAILURES ON SYSTEM LEVEL. (SS = SEMANTIC SIMILARITY, SSD = STATISTICAL SIGNIFICANT DIFFERENCE)

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

C. *RQ3: Comparing the Execution Time*

The mean execution time for all 10 executions of each technique is shown in Table V. Overall, Levenshtein was the least efficient technique in all projects analysed, followed by NCD and then Semantic Similarity. We had to interrupt the execution of Levenshtein for the Math project since it was taking many hours at each iteration. Since all techniques rely on a distance matrix to prioritise the tests, the main bottleneck lies in the distance measures. Levenshtein has polynomial complexity, whereas NCD depends on the compression time. Surprisingly, Semantic Similarity was more efficient than both algorithms. However, we do not account for the time to train the language model, as that is typically done only once.

RQ3: Generally, Jaccard is the fastest to execute and Levenshtein is the slowest. Our implementation of Semantic Similarity was faster than NCD.

VI. DISCUSSION

A. *Text of Requirements and Test Artefacts*

To begin with, Semantic Similarity was the only technique able to outperform Random on a system level. Therefore, the lexical approaches were not able to capture much of the meaningful requirements information from the test cases to convey unique requirements information. This suggests that Semantic Similarity –specifically Doc2Vec-based approaches– are just as effective in test prioritization, in comparison to other fields and areas within software engineering where such semantic diversity has been explored [5], [28], [31].

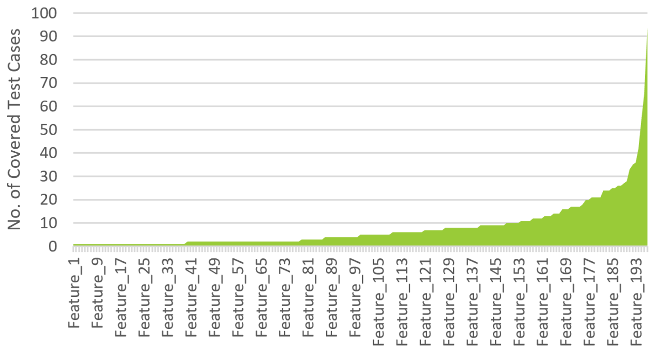

Fig. 6. Features/Requirements distribution among 1605 test cases for Company B. Note that only a subset of the 196 features is shown in the x-axis. Feature names are anonymised. The shape indicate a subset of features that are covered by a large number of tests (skewed to the right).

To mitigate risks with noisy data, we analysed the traceability between test cases and requirements. We found that the coverage of a prioritized test suite was influenced by the distribution of the features. Company A had 33% of the total test cases linked to at least one requirement (one-to-one), whereas the remaining tests were linked to more than one requirement (one-to-many). In contrast, all test cases from CompanyB were linked to exactly one requirement. Our close investigation of the text artefacts reveals some insights on why Random outperformed the lexical approaches (Jaccard, Levenshtein and NCD).

The techniques assume that tokens used in the requirements will also show up in the test cases. In Company B, the content of tests are not necessarily close 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 levels. Moreover, half of the features are covered by only 20% of the test suite (as shown by the histogram in Fig. 6), such that additional coverage can only be achieved by adding individual tests that exclusively cover a specific requirement. Since it performed better than Random, we argue that semantic similarity was able to capture those nuanced connections between test specifications and system requirements.

B. *Failure-Detection Rate*

We found that text-based diversity techniques were more effective than Random across all levels in failure detection (except Jaccard on the integration level). This is in line with Henard et al.'s findings regarding the acceptable amount of

TABLE V

EXECUTION TIME (IN MINUTES) OF THE DIFFERENT TECHNIQUES (MEAN AND STANDARD DEVIATION). SEMANTIC SIMILARITY WAS ONLY EXECUTED AT THE SYSTEM-LEVEL PROJECTS. (SS = SEMANTIC SIMILARITY)

Project	Test Suite Size	Jaccard	Lev	NCD	SS	Random
Comp A - System	2691	3.56 ± 0.29	481.95 ± 22.78	64.68 ± 1.92	20.53 ± 7.07	0.627 ± 0.049
Comp B - System	1605	0.77 ± 0.2	31.9 ± 3.49	14.29 ± 3.61	7.85 ± 1.99	0.204 ± 0.031
Comp A - Integ.	1605	0.91 ± 0.09	118.7 ± 45.05	19.34 ± 0.75	—	0.464 ± 0.079
Cli	262	0.32 ± 0.03	0.88 ± 0.51	0.56 ± 0.26	—	0.004 ± 0.002
Codec	440	0.76 ± 0.09	4.21 ± 2.09	1.19 ± 0.56	—	0.006 ± 0.002
Gson	988	0.49 ± 0.02	3.6 ± 0.45	3.19 ± 0.41	—	0.012 ± 0.001
JacksonCore	356	0.41 ± 0.04	2.21 ± 0.75	0.87 ± 0.35	—	0.005 ± 0.001
JxPath	347	0.4 ± 0.01	1.65 ± 0.26	0.75 ± 0.06	—	0.005 ± 0
Lang	1786	1.03 ± 0.23	33.58 ± 10.02	12 ± 3.74	—	0.022 ± 0.003
Math	2513	2.23 ± 1.29	—	30.49 ± 20.62	—	0.03 ± 0.011

faults even with restricted resources [2]. However, we did not see a difference between the techniques themselves. This indicates that the textual information in Java code does not seem to influence the choice betwee Jaccard, NCD or Levenshtein. Therefore, efficiency should be the deciding factor in which Jaccard is the most suitable choice.

While the integration and system level failure information are difficult to compare equally due to the scarcity of projects, the individual techniques have some visual differences. Similarly to the discussion on requirements coverage, we analysed the distribution of failures across different requirements and tests. The vast majority of failures (more than 900) were triggered by less than five requirements. 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. Similarly, Hemmati et al.'s results concluded that the best case scenario for diversity-based test selection in terms of effectiveness is when the tests that detect discrete faults are diverse [23]. Thus, researchers should contrast both criteria when analysing effectiveness.

C. Recommendations

Below, we condense the outcomes of this experiment into bulleted recommendations for practitioners. We also present some recommendations in Table VI regarding the application of specific diversity measures for different testing levels.

- 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. 6), then we recommend Semantic Similarity, as it provides the highest coverage over all system-level projects. For very limited time, we recommend Random at system-level or Jaccard at unit-level.
- If failure detection rate is the priority of the practitioner, we suggest to evaluate the history of previous failures beforehand to understand the nature of the failures distribution among test cases or requirements. 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 choosing a diversity-based 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, which significantly increases the execution time. Generally, we do *not* recommended using Levenshtein due to its expensive execution time and high variance.
- 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, particularly if using natural language.

VII. VALIDITY THREATS

In a fractional factorial experiment, several inherent limitations exist. As semantic-based diversity was not executed on integration level, we could not contrast the impact of natural language 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.

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, mitigating the effect of this limitation.

We also acknowledge the following limitations in our experiment executation. First, time constraints prevented us from executing Levenshtein on the Math project. Another threat to validity is the risk of test suite differences between different versions of the projects in Defects4J. On the other hand, executing only a subset of versions would skew results to the choice of version. The number of versions for each project ranged from 18 to 106. We mitigated this limitation by using

TABLE VI
RECOMMENDATIONS OF LEVELS OF TESTING TO USE GIVEN A TEXT-BASED DIVERSITY TECHNIQUE AND ITS TRADE-OFF.

Technique	**Trade-off**	**Recommended level**
Jaccard	Fast to execute on all levels. Bad coverage on system level. Good Failure detection on unit level but bad on system and integration levels.	Unit level
Levenshtein	Slow on all levels. Bad coverage on system level. Best Failure detection on integration level, worst Failure detection on unit level	Small test suite on Integration level.
NCD	Bad coverage on system level. Good Failure detection on all levels.	Unit/ System level
Semantic Similarity	Faster than NCD and Levenshtein. Good coverage, Moderate Failure detection rate. Can only be applied on system level.	System level
Random	Fast to execute on all levels. Good coverage. Bad failure detection.	When $\leqslant 30\%$ or $\geqslant 90\%$ budget

projects of differing sizes and versions to show more variety in our results. 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. Our data did not show large variances in the sample that hinder the reliability of our statistical or visual analysis.

The need to parallelise the execution of the experiment 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.

VIII. CONCLUSION

Our paper compares four text-based diversity prioritization techniques 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. Our experiment shows that semantic-based diversity can lead to better requirements coverage when compared to lexical diversity depending on the budget allocated for test execution (ideally greater than 20%). In contrast, all lexical diversity techniques had similar results on failure coverage when applied to unit-level tests. For integration-level tests, Levenshtein and NCD were able to provide significantly better failure coverage by executing between 20–50% of the test suite. We summarise our results into recommendations to practitioners by contrasting the trade-off of each type of diversity measure applied to different levels of tests.

Future research aims to explore further the trade-off of Semantic Similarity on more system-level tests as well as unit and integration levels. Our goal is to use large language models on mixed artefacts that include both natural language and code. We also aim to expand our automated test prioritisation pipeline with interactivity to recommend the percentage of tests that should be executed (budget) given a summary of previous executions of the prioritised test suite.

ACKNOWLEDGEMENT

This work was partially supported by the Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by Knut and Alice Wallenberg Foundations.

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] 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.

[3] 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.

[4] 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.

[5] 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.

[6] S. Tahvili, L. Hatvani, E. Ramentol, R. Pimentel, W. Afzal, and F. Herrera, “A novel methodology to classify test cases using natural language processing and imbalanced learning,” *Engineering applications of artificial intelligence*, vol. 95, p. 103878, 2020.

[7] 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.

[8] 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.

[9] 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.

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

[11] A. M. Dai, C. Olah, and Q. V. Le, “Document embedding with paragraph vectors,” *arXiv preprint arXiv:1507.07998*, 2015.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, *et al.*, "Language models are few-shot learners," *Advances in neural information processing systems*, vol. 33, pp. 1877–1901, 2020.

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

[19] 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.

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

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

[22] 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.

[23] 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.

[24] 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.

[25] C. Landing, S. Tahvili, H. Haggren, M. Langkvis, A. Muhammad, and A. Loufi, "Cluster-based parallel testing using semantic analysis," in *2020 IEEE International Conference On Artificial Intelligence Testing (AITest)*, pp. 99–106, 2020.

[26] H. Aman, T. Nakano, H. Ogasawara, and M. Kawahara, "A test case recommendation method based on morphological analysis, clustering and the mahalanobis-taguchi method," in *2017 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pp. 29–35, IEEE, 2017.

[27] Y. Yang, X. Huang, X. Hao, Z. Liu, and Z. Chen, "An industrial study of natural language processing based test case prioritization," in *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, pp. 548–549, IEEE, 2017.

[28] 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.

[29] 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.

[30] 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.

[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.