Automatically Identifying Shared Root Causes of Test Breakages in SAP HANA2

Gabin An*
School of Computing, KAIST
Daejeon, Republic of Korea
agb94@kaist.ac.kr

Juyeon Yoon*
School of Computing, KAIST
Daejeon, Republic of Korea
juyeon.yoon@kaist.ac.kr

Jeongju Sohn
SnT, University of Luxembourg
Luxembourg
jeongju.sohn@uni.lu

Jingun Hong
SAP Labs Korea
Seoul, Republic of Korea
jingun.hong@sap.com

Dongwon Hwang
SAP Labs Korea
Seoul, Republic of Korea
dong.won.hwang@sap.com

Shin Yoo
School of Computing, KAIST
Daejeon, Republic of Korea
shin.yoo@kaist.ac.kr

ABSTRACT

Continuous Integration (CI) of a large scale software system such as SAP HANA2 can produce a non-trivial number of test breakages. Each breakage that newly occurs from daily runs needs to be manually inspected, triaged, and eventually assigned to developers for debugging. However, not all new breakages are unique, as some test breakages would share the same root cause; in addition, human errors can produce duplicate bug tickets for the same root cause. An automated identification of breakages with shared root causes will be able to significantly reduce the cost of the (typically manual) post-breakage steps. This paper investigates multiple similarity functions between test breakages to assist and automate the identification of test breakages that are caused by the same root cause. We consider multiple information sources, such as static (i.e., the code itself), historical (i.e., whether the test results have changed in a similar way in the past), as well as dynamic (i.e., whether the coverage of test cases are similar to each other), for the purpose of such automation. We evaluate a total of 27 individual similarity functions, using real world CI data of SAP HANA2 from a six month period. Further, using these individual similarity functions as input features, we construct a classification model that can predict whether two test breakages share the same root cause or not. When trained using ground truth labels extracted from the issue tracker of SAP HANA2, our model achieves an F1 score of 0.743 when evaluated using a set of unseen test breakages collected over three months. Our results show that a classification model based on test similarity functions can successfully support the bug triage stage of a CI pipeline.

KEYWORDS

Continuous Integration, Test Similarity, Root Cause Analysis

*These authors contributed equally to this research.

Woodstock '18, June 03–05, 2018, Woodstock, NY

ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/10.1145/1122445.1122456

ACM Reference Format:
Gabin An, Juyeon Yoon, Jeongju Sohn, Jingun Hong, Dongwon Hwang, and Shin Yoo. 2018. Automatically Identifying Shared Root Causes of Test Breakages in SAP HANA2. In *Woodstock '18: ACM Symposium on Neural Gaze Detection, June 03–05, 2018, Woodstock, NY*. ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/1122445.1122456

1 INTRODUCTION

SAP HANA2 is a large scale in-memory, relational database management system, which has a large codebase that, as of October 2021, consists of more than 11MLoC across about 50K files that are mostly written in C and C++. Its Continuous Integration (CI) system performs post-submit testing [10] for the main branch on a daily basis (hereafter, referred to as *daily test run*), in order to test all code changes submitted during the day together. In each daily test run, more than 3,000 test cases[1] are chosen to be executed; some of the executed tests would fail, either due to newly introduced faults, or residual faults that have been revealed in earlier iterations but are yet to be fixed. When a test case fails, the CI system re-executes the test case three times to cater for test flakiness. If all three retest attempts result in failures, the original failure is considered as a *test breakage*; otherwise, it is classified as a flaky test [32].

Based on the test history collected in 2021, about 30 test breakages are produced by each daily test run on average: a single breakage takes 6.5 days on average to fix. Part of the time-to-fix cost is due to the test review process that follows every test breakage. During the review, a human developer has to interpret the test result to identify the root cause [37]. If the root cause is thought to be a known one, the test that resulted in the breakage is added to the corresponding and existing bug ticket; if it is considered a new one, a new bug ticket is created and assigned to the corresponding test. The bug can be assigned to the responsible developers only after a bug ticket is assigned to the broken test case.

The review process is not fully automated and involves manual interpretation and analysis of the test results. Some of the automated sub-tasks are costly as well: for example, each bug ticket initiates the bisection of the repository history using the broken test cases in order to identify the bug introducing change. Many test cases used in the post-submit testing stage are integration tests

[1]In SAP HANA2, a test case refers to a test script that contains numerous atomic test methods.

with high initialisation and set-up costs. Consequently, bisection based on these tests can also be extremely costly.

Given the current CI workflow of SAP HANA2, any redundant bug tickets would incur a large amount of unnecessary review cost, both in terms of human effort (inspection and analysis) and computational resource (bisection). However, redundant bug tickets do occur. It is expected that a single bug can cause the breakages of multiple test cases, so the mapping between bugs and test breakages is clearly not one-to-one. Since the workflow currently rely on manual inspection and analysis to identify the root cause, there is room for human error, leading to duplicate and redundant bug tickets: the issue tracker of SAP HANA2 actually anticipates this, and allows duplicate bug tickets to be marked by developers.

To improve the efficiency of the overall CI pipeline, it would help greatly if we can reduce the number of bug tickets generated, while not losing any diagnostic capability of the post-submit test stage. In this paper, we study a set of test similarity functions to evaluate whether they can serve as a reliable indicator of shared root causes. The similarity functions we study are based on a range of information sources: static similarity based on source code similarity, historical similarity based on the accumulative history of test result changes, and dynamic similarity based on the similarities between coverages achieved by test cases. First, using various configurations, we evaluate a total of 27 similarity functions individually: our results show that different similarity functions complement each other. Based on this observation, we subsequently train a classification model using these similarity functions to produce input features. An empirical evaluation of our classification model using real CI test results of SAP HANA2 shows that it can classify whether a pair of test cases share the same root cause or not with F1 score of up to 0.743. We plan to integrate the studied similarity functions and classification models into the CI pipeline of SAP HANA2.

The rest of the paper is organised as follows. Section 2 introduces the similarity functions we study. Section 3 describes how we construct a classification model using the input features obtained with the similarity functions. Section 4 presents the configuration of our empirical evaluation, whose results are discussed in Section 5. Section 6 discusses threats to validity, and Section 7 presents related work. Finally, Section 8 concludes.

Table 1: Example of how different tokenisers process the identifier `test_ldap_sessionfactory`

Tokeniser	Tokenisation Result
Elementary	`['test', 'ldap', 'sessionfactory']`
Ronin	`['test', 'ldap', 'session', 'factory']`

2 MEASURING SIMILARITY BETWEEN TEST BREAKAGES

Given two test breakages, we assume that the similarity between the two corresponding test cases is a good indicator of whether the breakages share the same root cause or not. There are various information sources that can be used to measure the similarity between test cases. We broadly categorise the information sources we use in this work into *static*, *historical*, and *dynamic*. This section will describe each of the information sources and their corresponding similarity functions in details.

2.1 Static Information

Static information is what can be obtained without running the target program, e.g., the data from source code or the structure of the software. In particular, we focus on *the name of the test case* as it can directly reflect the purpose and the intention of the test case [7]. Suppose two test cases called `testB*_O***_C***_2_FLAT_SQL` and `testB*_O***_C***_2_CLASSIC_SQL` break: we may suspect that they share the same root cause, because their names are similar to each other (names are partially masked for confidentiality). Consequently, we use the string similarity between the names of two test cases as a proxy for the similarity between their breakages. There are numerous string similarity measures that quantify the lexical similarity between two strings. We use the widely studied Jaro-Winkler [41], which is a type of normalised edit distance, as our baseline, because it is purely lexical and does not reflect any semantic intention. We do not use Levenshtein distance as it cannot be easily normalised [27].

Our primary choice of similarity measure between test case names is a token-based approach, which first tokenise the given strings and use the frequencies of tokens (sometimes also their orders) to measure the similarity between the strings. The motivation behind our choice is that test case names are typically composite words that consist of multiple tokens, each reflecting either the test purpose or the test target (e.g., `test_ldap_sessionfactory`). There are various methods for tokenising source code identifiers that have been suggested in literature [5, 11, 15, 21, 25]. Table 1 presents the example of tokenisation results using different source code tokenisers in `Spiral` [21], a package that implements a range of tokenisation algorithms for identifiers in source code. Ronin, the most advanced tokeniser provided by `Spiral`, is based on Samurai [11] tokenisation algorithm that regards token frequencies mined from public source code repositories. By adding various heuristic rules, Ronin recognise the terms in an identifier more accurately than the elementary tokeniser. We refer to the set of all resulting tokens as the *vocabulary*, V.

After tokenisation, we vectorise the tokens for the similarity computation. We use the widely-adopted Count (Term Frequency) and TF-IDF (Term Frequency-Inverse Document Frequency) vectorisers [9, 35]. A test case name corresponds to a vector, whose length is equal to the size of the vocabulary. For the token $t_i \in V$, the corresponding element in the vector representation, e_i, is either the token occurrence frequency (by the Count vectoriser), or the token occurrence frequency times inverse document frequency (by the TF-IDF vectoriser). The inverse document frequency is usually defined as:

$$idf(t) = log\frac{1+n}{1+df(t)} + 1$$

where $df(t)$ is the number of documents (here, test cases names) that contain the token t. The less frequently the term t occurs, the higher the $idf(t)$ value becomes. The TF-IDF vectoriser consequently gives lower weights to the more common tokens, such as `'test'`, a commonly used prefix among test case names. Once we vectorise the test case names, we measure the similarity between

Table 2: Example of converting a 7-day test execution history into four types of vectors introduced by Golagha et al. [13]. The three different values for the test result, 'P', 'F', and 'N', refers to 'Passed', 'Failed', and 'Not executed', respectively.

Day	1	2	3	4	5	6	7
Test result of t	P	F	P	N	P	P	F
Failed vector, $v_f(t)$	0	1	0	0	0	0	1
Passed vector, $v_p(t)$	1	0	1	0	1	1	0
Broken vector, $v_b(t)$	0	1	0	0	0	0	1
Repaired vector, $v_r(t)$	0	0	1	0	0	0	0

them using Cosine Similarity, a widely-used text vector similarity metric [36].

2.2 Historical Information

Recently, Golagha et al. [13] proposed a method to measure the distance between test cases by exploiting the historical information of the test cases. We assume that, if two test cases are similar to each other in their purposes and intentions, they may have been not only broken due to the same reason, but also repaired by the same patch, in the past. Consequently, we also assume that the more similar the histories of the two tests are, the higher the probability of them sharing the same root cause is.

Following Golagha et al. [13], we first collect the execution history of test cases for the daily test runs within a specific time window (hereby referred to as *history collection period*). Subsequently, we construct the Failed, Passed, Broken and Repaired vectors (v_f, v_p, v_b, v_r) based on the trajectories of the test results. Failed and Passed vectors contain 1 whenever the corresponding previous test results are failed (F) and passed (P), respectively, and 0 otherwise. In comparison, Broken and Repaired vectors represent the transitions in test results: the Broken vector contains 1 whenever the corresponding previous test result is the result of a transition from pass to fail, and 0 otherwise. Similarly, Repaired vector contains 1 whenever the corresponding previous test result is the result of a transition from fail to pass, and 0 otherwise. Table 2 shows an example of these historical vectors based on how test case t changed its results. Consider the Broken vector as an example: its element for Day 2 is 1, because the t changed from P (Day 1) to F (Day 2).

Once we extract these vectors from the given test history, we can compute the similarity between two test cases based on each of these vectors. We actually consider two types of similarity: one is the average of Failed and Passed vector similarities between two test cases (s_{fp}), and the other is the average of Broken and Repair vector similarities between the two test cases (s_{br}), where *sim* can be any similarity metric between binary vectors:

$$s_{fp}(t,t') = \frac{1}{2} \cdot sim(v_f(t), v_f(t')) + \frac{1}{2} \cdot sim(v_p(t), v_p(t')) \quad (1)$$

$$s_{br}(t,t') = \frac{1}{2} \cdot sim(v_b(t), v_b(t')) + \frac{1}{2} \cdot sim(v_r(t), v_r(t')) \quad (2)$$

2.3 Dynamic Information

Following many existing studies that aim to measure similarities between test cases (such as failure clustering [12, 22, 34] or identification of coincidental correctness [28, 40]), we use code coverage to measure the similarity between two test breakages. Our rationale is that, if the coverages are similar, two test cases must have a similar intention and, consequently, are more likely to fail due to the same root cause.

Typically, the distance between test case coverages is measured by computing the distance between the *binary coverage vectors* [14, 20, 39, 43] using set- or vector-distance metrics such as Jaccard, Cosine, or Hamming. However, by representing each test case as an individual vector or set, we lose the relative importance of each program element. Given a binary coverage vector for a single failing test case, we can only assume that each covered element may be equally contributing to the breakage. It is only when we also consider the coverages of passing tests that the relative importance of each covered elements becomes clearer (i.e., those also covered frequently by passing test cases are less important when measuring similarities between test breakages).

There are existing works that try to incorporate information from passing test cases by considering the results of fault localisation when measuring similarities between test breakages [12, 22, 31]. Here, a test breakage is represented by the ranking of all program elements according to their suspiciousness with respect to the breakage [12]: if two test breakages share the same root cause, they will also result in a similar distribution of suspiciousness scores across the program. While this approach successfully incorporates the information from passing test cases into the similarity between test breakages, both the fault localisation and computation of distances between rankings (via the Kendall-Tau distance [26]) turned out to be too expensive to apply to SAP HANA2. Our previous work [1] showed that it takes several hours to compute the pairwise rank distances between a few dozen failing test cases from a project with about 40KLoC, rendering the approach impractical for our purpose.

To better capture the similarity between test breakage by incorporating information from passing test cases, we turn to our previous work [1] that proposed a novel test similarity measure based on the *hypergraph* modelling of coverage data: our results showed that failure clustering using hypergraph-based test distance is more accurate than those using other test distance functions, while being much more computationally efficient. A hypergraph refers to a graph whose edges can join any number of nodes, instead of only two (these edges are called *hyperedges*) [2, 44]. We can convert a coverage matrix into a hypergraph without any loss of information, where each node corresponds to a test case, and each edge, that represents a program element, joins test cases that executed the program element. Figure 1 shows an example of converting a coverage matrix into a hypergraph.

In a hypergraph, the linkage (similarity) between two nodes t and t' is usually defined as follows:

$$link(t,t') = \sum_{e \in E_t \bigcap E_{t'}} \frac{w(e)}{deg(e)} \quad (3)$$

where E_t is a set of hyperedges that join t, and $w(e)$ and $deg(e)$ are the predefined weight value and the degree of the hyperedge e,

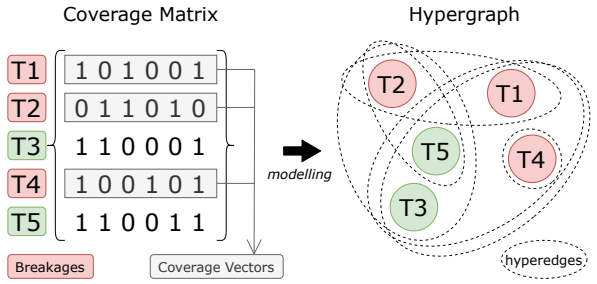

Figure 1: Example of converting a coverage matrix with five test cases (three breakages) and six program elements into a hypergraph with five nodes and six hyperedges

i.e., the number of nodes that e joins, respectively. Assuming that all hyperedges have equal weights, 1.0, the linkage between two nodes are the sum of the *reciprocal* degrees of hyperedges that join both of the nodes. Consequently, a higher degree of an edge leads to a lower contribution to the linkage value, i.e., more commonly covered program elements have less impact on the similarity between test coverages. Since the hypergraph-based test similarity uses the coverage of passing test cases as well as the failing test cases, it can reduce the impact that program elements commonly covered by passing test cases have on the test similarity, unlike the set and vector-based similarity metrics that only consider coverages of failing test cases. In our study, we use the normalised version of the node linkage value, nlink, as a similarity between test breakages:

$$nlink(t, t') = \frac{1}{2}\left(\frac{link(t, t')}{link(t, t)} + \frac{link(t, t')}{link(t', t')}\right) \quad (4)$$

If two tests t and t' do not share any program elements in their coverages, the $nlink(t, t')$ value becomes 0.0; if two tests have exactly the same coverage, the $nlink(t, t')$ value becomes 1.0.

Another benefit of hypergraph-based test similarity in the industrial context is its computational efficiency: it takes only about 1.0 second on average to compute the pairwise similarity between every test breakages from a daily test run of SAP HANA2. Please refer to our previous work regarding the hypergraph modelling, linkage computation and normalisation [1].

3 LEARNING AN UNIFIED CLASSIFIER

Section 2 described similarity functions for test breakages based on three different information sources: static, historical, and dynamic. Since these information sources are mutually exclusive, we argue that using all information sources together will help identify shared root causes of test breakages better, when compared to using a single information source. To use multiple similarity functions together, we construct a binary classification model that takes multiple similarity functions between two test breakages as input features and predicts whether they share the same root cause.

We use a basic Multi-layer Perceptron (MLP), shown in Figure 2, as our classification model. Given a set of test breakage pairs, we train the MLP model using features obtained from the similarity functions and the ground-truth relevance labels. A relevance label indicates whether two tests share the same root cause or not, and is

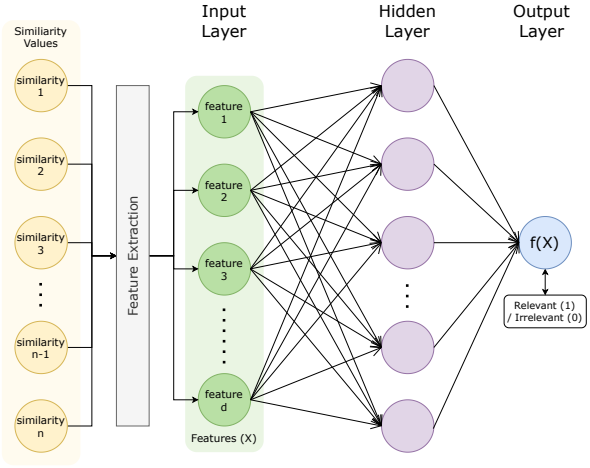

Figure 2: A MLP Classifier with one hidden layer that predicts the relevance of test pairs using input similarities

retrieved from the existing bug ticket information (see Section 4.2.1 for the way ground truths are collected).

4 EXPERIMENTAL SETUP ON SAP HANA2

We evaluate the performance of the similarity functions and the classification model in terms of predicting the relevance between test breakages in SAP HANA2. This section presents the research questions and describes our experimental settings including the evaluation dataset, the similarity function configurations, the details of our classification model, and the evaluation metrics.

4.1 Research Questions

We ask the following three research questions:

- **RQ1. Similarity Function Effectiveness**: How effective is each similarity function when used to classify the relevance of test breakage pairs?
- **RQ2. Similarity Function Uniqueness**: How complementary are the similarity functions to each other?
- **RQ3. Model Effectiveness**: How effective is the unified model incorporating all information sources when compared to using only a single similarity function?

4.2 Evaluation Dataset Construction

From SAP HANA2 CI, we mine test result trajectories from the period of six months, February to July 2021. For each daily test run, we consider all pairs of test breakages from the run and predict whether they share the same root cause or not. The following subsections will describe the ground-truth collection procedure and the construction of training/test datasets.

4.2.1 Collecting Ground-Truth Labels. To evaluate our classification model, we need to establish the ground truth relationships between test breakages, i.e., the pairs that actually share the same root cause. We use the information recorded with the bug tickets to establish ground truth relationships between test pairs. A bug ticket is manually generated to represent a single, unique bug that is yet

Table 3: Statistics of the Evaluation Dataset

Dataset	Month	# total pairs	# relevant pairs	% relevant
Training	2	7,922	149	1.88%
	3	3,806	72	1.89%
	4	8,005	298	3.72%
	Total	19,733	519	2.63%
Test	5	6,767	126	1.86%
	6	6,621	61	0.92%
	7	4,205	25	0.59%
	Total	17,593	212	1.21%

to be resolved, and is assigned to multiple tests whose breakages share the same root cause. Consequently, it represents the human interpretation of the test results as well as the mapping between root causes and test breakages.

A bug ticket can be assigned to a test in two different scenarios. One is when the developers assign one of the existing tickets that represent known yet unresolved bugs, to the test that resulted in a newly observed breakage. The other is when a test breakage reveals a new bug, forcing the developer to issue a new bug ticket and assign it to the test case that resulted in the breakage.

Since we only have a mapping between bug tickets and test cases, however, it may not be clear which actual breakages are connected to a specific bug ticket. We use the following heuristic to determine whether a bug ticket t is linked to a specific test breakage b:

- A bug ticket t already exists, is assigned to a test case c, and remain unresolved when c results in a breakage b, or
- A bug ticket t is newly created and assigned to test case c within seven days of the breakage b of test c

The seven-day window reflects the maximum duration typically required to assign bug tickets to newly occurring test breakages in the SAP HANA2 CI workflow. Although each bug ticket should ideally represent a single unique bug, it is also possible that bug tickets are duplicated (i.e., created multiple times for the same bug). This is because bug duplicate detection is currently performed manually and, therefore, is susceptible to human error. In case a bug ticket is identified as a duplicate by a human developer, it is marked as such on the issue tracker: we consider all duplicates as the same bug ticket based on this marking.

4.2.2 Splitting Training/Test Datasets. The collected six-month test history data is divided into training and test datasets. The data from the preceding three months (from February to April) are used as training data, while the data of the remaining three months (from May to July) are used as test data for the performance evaluation of unseen data. Table 3 shows the summary of dataset statistics.

4.3 Configuration and Implementation Details

This section describes the experimental settings, implementation details for the similarity functions, and the classification model.

4.3.1 Similarity Functions. Table 4 shows the possible configuration settings for each information source in our experiment. The 'Component' and 'Values' columns denote the name and the possible values of each configurable component, respectively.

Table 4: Configuration components of a similarity function for each information source

Source	Component	Values
Static (Token)	Tokeniser	Elementary, Ronin
	Vectoriser	Count, TF-IDF
	Similarity Measure	Cosine
Static (Edit)	Similarity Measure	Jaro-Winkler
Historical	Collection Period	90, 180, 365 days
(Failed/Passed)	Similarity Measure	Cosine, Hamming, Jaccard
Historical	Collection Period	90, 180, 365 days
(Broken/Repaired)	Similarity Measure	Cosine, Hamming, Jaccard
Dynamic	Similarity Measure	Cosine, Hamming, Jaccard, nlink [1]

For the token-based static information source, we can vary the tokeniser and the vectoriser, for which we use the open-source implementations of `Spiral`[2] and `sklearn-0.24.1`, respectively. For Jaro-Winkler string similarity, we use the implementation from an open-source Python library `jellyfish` [3] that provides a set of string comparison algorithms.

For the historical information source, we can vary the duration of the historical time window, as well as the similarity metric *sim* used by s_{fp} and s_{br}, each defined in Equation 1 and 2 in Section 2.2. We use three window lengths of 90, 180, and 365 days to evaluate its impact on classification performance. Since historical vectors are binary, we employ the Hamming and Jaccard similarity metrics in addition to the Cosine similarity metric as the vector similarity metric *sim*, and use the implementation provided by `scipy-1.4.1`.

For the dynamic information source, we use the hypergraph-based similarity score, *nlink*, in addition to the three baselines, Cosine, Hamming, and Jaccard, that can compute the similarity between two binary coverage vectors. Note that, to save the cost of daily test runs, the CI pipeline of SAP HANA2 collects coverage weekly rather than daily. Therefore, we use the most recently measured coverage data for each test case when needed. Throughout this paper, we use a file-level coverage matrix that contains information about which test cases execute which files.

Overall, we use a total of 27 different similarity functions ($2 \cdot 2 + 1$ from Static, $2 \cdot 3 \cdot 3$ from Historical, and 4 from Dynamic) to evaluate all configuration combinations from each information source.

4.3.2 A Classification Model. Our base MLP classification model is a vanilla MLP classifier [18] that contains one fully-connected hidden layer with 100 neurons: we use the implementation from `sklearn-0.24.1` with the default parameter setting. We train the model for the maximum of 300 epochs with the learning rate of 0.001: our early-stopping criterion is to stop when there is no improvement over ten consecutive epochs.

In addition, to reduce the chance of overfitting, we also use a *bagging* (bootstrap aggregating) [4] ensemble model that aggregates the prediction output from n different base estimators trained independently from each other. Specifically, we use a bagging-MLP model consisting of ten vanilla MLP classifiers, each trained with

[2] https://github.com/casics/spiral
[3] https://github.com/jamesturk/jellyfish

random subsets of the samples and features drawn from the original training dataset with replacement: we bootstrap until we have the same number of samples and features as the original data. To minimise the effect of randomness, we train a total of ten models with different random seeds for both the single MLP and bagging-MLP models, and report the average and the standard deviation.

We use the following feature selection and extraction schemes [16]:

(1) ***All Features*** uses all available similarity functions from each information source, i.e., a total of 27 similarity functions.
(2) ***Best Features*** uses only the best performing configuration of the similarity function for each information source. Since there are two types of historical similarity, Failed/Passed and Broken/Repaired, we use the best performing configuration for each. Overall, four similarity values are used: one from Static, two from Historical, and one from Dynamic.
(3) ***PCA Features*** uses the d-dimensional features extracted using Principal Components Analysis (PCA) [42]. PCA is an widely-used *dimensionality reduction* technique that reduces the number of features while retaining maximum information. It computes the principal components of features and uses them as a new basis for the feature space. We set d to $\{4, 10, 16, 22, 27\}$, where 4 is equal to the number of the *best* features, and 27 is equal to the number of *all* features.

4.4 Evaluation Metrics

We use commonly-used evaluation metrics for binary classification [19] such as *precision*, *recall*, and *F1* (the harmonic mean of precision and recall). We also use the following two advanced evaluation metrics that summarise the Receiver Operating Characteristic (ROC) and Precision-Recall (PR) curves for a range of threshold values into a single score [3, 8]:

- **AUROC** (Area Under the ROC Curve): An ROC curve is a graph showing the performance of a classification model at all classification thresholds in terms of True Positive Rate and False Positive Rate[4]. In this study, AUROC measures the area underneath the ROC curve from (0,0) to (1,1), as all similarity values are normalised. The baseline for AUROC is 0.5 of a random binary classifier.
- **AP** (Average Precision): AP is the weighted mean of precision values at each threshold where the weight value is set to the increase in recall from the previous threshold. Formally, it is defined as

$$AP = \sum_{n} (R_n - R_{n-1}) P_n$$

where P_n and R_n are the precision and recall value at the nth threshold, respectively. Note that AP is one of the methods for calculating the area under the PR curve (AUPRC). The baseline for AP is the proportion of positive samples.

While AUROC is the same regardless of what the "positive" class is, the AP value indicates how correctly a model handles positive samples. As such, it may be more useful in our context, because the proportion of positive samples (i.e., relevant test pairs) is significantly lower than that of negative samples. This is important because we will report the samples predicted as positive to the human engineers.

[4]TPR = TP/(TP+FN) and FPR = FN/(FN+TP)

Table 5: Prediction performance of each similarity function on the training data. The baseline values of AUROC and AP are 0.5 and 0.024, respectively. (F/P: Failed/Passed, B/R: Broken/Repaired)

Source	Similarity function configuration		AUROC (B: 0.500)	AP (B: 0.026)
	Tokeniser	*Vectoriser*		
Static (Token)	Elementary	Count	0.913	0.673
	Elementary	TF-IDF	0.909	0.694
	Ronin	Count	**0.918**	0.680
	Ronin	**TF-IDF**	0.916	**0.706**
Static (Edit)	*Measure*			
	Jaro-Winkler		0.894	0.578
	Measure	*Period*		
Historical (F/P)	Cosine	90 days	0.734	0.157
	Cosine	180 days	0.731	0.163
	Cosine	**365 days**	0.808	**0.206**
	Hamming	90 days	0.811	0.149
	Hamming	180 days	0.784	0.129
	Hamming	365 days	0.724	0.081
	Jaccard	90 days	**0.814**	0.154
	Jaccard	180 days	0.809	0.169
	Jaccard	365 days	0.786	0.185
	Measure	*Period*		
Historical (B/R)	Cosine	90 days	0.708	0.256
	Cosine	**180 days**	0.792	**0.367**
	Cosine	365 days	0.871	0.362
	Hamming	90 days	0.695	0.074
	Hamming	180 days	0.694	0.070
	Hamming	365 days	0.714	0.053
	Jaccard	90 days	0.860	0.115
	Jaccard	180 days	**0.903**	0.208
	Jaccard	365 days	0.875	0.294
Coverage	*Measure*			
	Cosine		0.866	0.545
	Hamming		0.844	0.326
	Jaccard		0.864	0.539
	nlink (Eq. 4)		**0.960**	**0.740**

5 EVALUATION RESULTS AND ANALYSIS

This section analyses the results of our empirical evaluation.

5.1 RQ1: Effectiveness of Similarity Functions

Table 5 presents the AUROC and AP scores obtained by all similarity functions against our training data. All similarity functions have higher scores than the baseline scores of AUROC and AP, 0.5 and 0.026, respectively. Among them, the coverage-based similarity

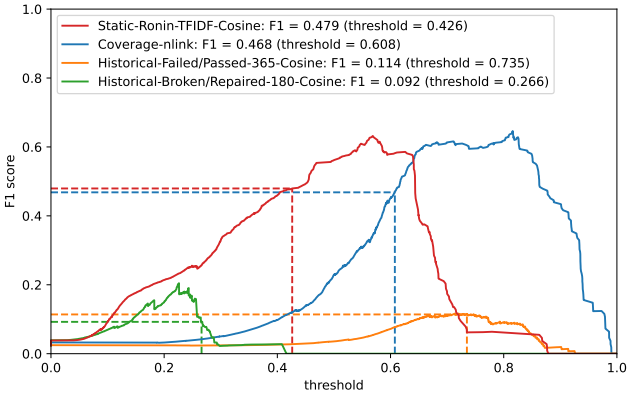

Figure 3: F1 scores of each similarity function at different thresholds (calculated on the test data)

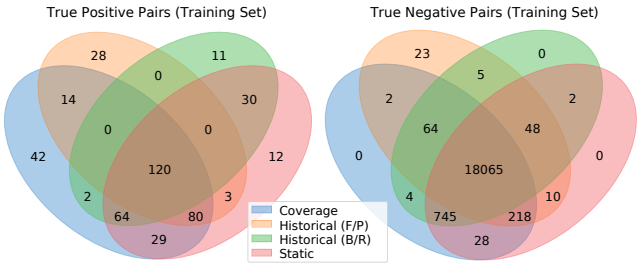

Figure 4: Venn diagram of True Positive (TP) and True Negative (TN) samples on the training data with the best-performing similarity functions selected based on the AP values (the threshold with the maximum F1 score is used for each similarity function)

function, nlink [1], significantly outperforms all other similarity functions in terms of both AUROC and AP. In comparison, the other coverage-based similarity functions, Cosine, Hamming, and Jaccard, all show poorer classification performance than the static-based similarity functions as well as nlink.

The AP values show that static information can identify the relevant pairs more accurately than historical information. When measuring the similarity between the test names, the token-based similarity functions outperform Jaro-Winkler, an edit-distance-based baseline. Especially, the more advanced tokeniser, Ronin, combined with the TF-IDF term-weighting scheme, achieves the highest AP value. Among the historical similarity functions, those with the Cosine and Jaccard similarity metrics perform better than those with Hamming. When measuring failed/passed similarity, the longer the collection period is, the better the performance becomes in general, except for broken/repaired similarity for which no such correlation is observed.

From these evaluation results on the training data, we choose the best performing similarity function with the highest AP value from each information source. According to the previous work [13] that first proposed these history-based distance metrics, Failed/Passed and Broken/ Repaired capture different characteristics. Thus, we decide to select one from each category for the following experiments. The four selected similarity functions that represent each category are highlighted with a grey background colour in Table 5. Using the training data, we learn the *optimal* similarity threshold for each selected similarity function. We use the widely used method of choosing a threshold with *the maximum F1 score*, i.e., the threshold that achieves the best balance between precision and recall [29]. For example, nlink has the maximum F1 score, 0.733, at the threshold of 0.608 on the training data.

We evaluate the four selected functions on the test data: Figure 3 shows the F1 scores of each function at different thresholds against the test data. Each dotted line represents the F1 score at the learnt threshold on the training data, where the values are also shown in the graph legend. The results show that the maximum F1 score of nlink, 0.646, is higher than that of other similarity functions. However, when using the learnt threshold, the static and coverage similarity functions shows almost the same F1 scores, 0.479 and 0.468. From the graph, we can observe a considerable discrepancy between the learnt threshold and the best-performing threshold on the test data, except for the passed/failed similarity function. This motivates us to develop a more elaborated classification model that better generalises to unseen data.

Answer to RQ1: The coverage-base similarity function, nlink, significantly outperforms all other similarity functions, achieving AUROC of 0.960 and AP of 0.740, when predicting the relevance between test breakages. Based on the best-performing similarity function from each information source, the relative effectiveness of each information source is Coverage > Static > Historical (B/R) > Historical (F/P).

5.2 RQ2: Uniqueness of Similarity Functions

Venn Diagram in Figure 4 shows True Positive (TP) and True Negative (TN) pairs in the training data that each similarity function correctly classifies with its optimal threshold. In terms of TP, we can see that certain pairs are correctly classified by only a specific similarity function and nothing else. For example, using the coverage similarity (nlink) enables us to find 42 relevant pairs that are found by no other functions. A concrete example is shown in Figure 5: given the test breakage (in the red rectangle box) from a daily test run, if we rank all other breakages in the test run in the descending order of their nlink similarities to the given breakage, the top-ranked test case actually shares the same bug ticket with the target breakage and is classified as relevant with the optimal threshold of nlink, which is 0.608. Since those two test case names have no common tokens (except for `"test"`), it would be difficult for static similarity functions to capture the similarity.

While the coverage-based similarity function, nlink, correctly classifies the most TP samples uniquely, the TN results show that only the Failed/Passed historical similarity function could correctly identify irrelevant pairs that no one else could find, despite producing the lowest AP values among the four functions.

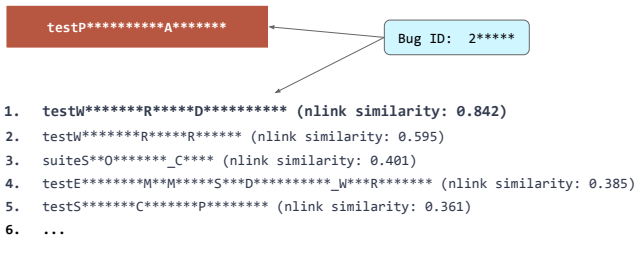

Figure 5: An example of a test breakage pair that is correctly classified by only the coverage-based similarity function, nlink. Due to confidentiality issues, the test names are partially masked. The unmasked character represents the first character of each token.

Answer to RQ2: The results of the TP and TN samples show that none of the similarity functions is completely dominated by other functions, which means that the similarity functions from different information sources can complement each other. Together with the results of RQ1, our findings for RQ2 clearly demonstrate the need for an unified classification model that incorporates all sources of information.

5.3 RQ3: Effectiveness of Classification Model

Figure 6 shows the F1 scores of the MLP-based unified models for different feature extraction strategies (Section 4.3.2) against the test data. For each feature strategy, there are two bars, each showing the performance of the MLP classifier with, and without, bagging, respectively. For easier comparison with the previous results, Figure 6 also shows the F1 scores of the two best performing similarity functions, taken from Figure 4, with horizontal lines.

Among the studied models, the bagging ensemble model with PCA feature extraction ($d = 16$) performs the best, achieving an F1 score of 0.743: this is at least 55% improvement over the highest F1 score achieved by a single similarity function when using the learnt threshold, 0.479 (as shown in Figure 3). Note that the F1 score of 0.743 achieved by the bagging ensemble model is also higher than the highest F1 score achieved by a single similarity function at any threshold, which is 0.646. Overall, for almost all feature extraction strategies, the bagging ensemble model shows better performance than the single MLP model in terms of the F1 scores. The results show that the use of multiple base estimators can reduce the potential for overfitting. Moreover, the narrower confidence intervals of the bagging models suggest that their performance is relatively stable than that of single models.

In terms of the feature extraction methods, the PCA Features strategy tends to outperform all other strategies. This is because PCA reduces overfitting so that the model can generalize better. Our results show that reducing the dimensionality by a moderate amount, from 27 to 16, can effectively reduce the noise in the training data without losing much information. However, since more dimensionality reduction naturally leads to more information loss, PCA with the smallest feature number, 4, performs worse than the PCA with the larger feature number, as well as the All Features strategy.

Table 6: Mean (standard deviation in parentheses) of precision and recall for each feature extraction strategy, with or without bagging. The highest precision and recall for MLP and Bagging_MLP are typeset in bold respectively.

Features	MLP		Bagging_MLP	
	Precision	Recall	Precision	Recall
Best	0.630 (0.047)	0.514 (0.023)	0.613 (0.019)	0.594 (0.005)
PCA_4	**0.829** (0.076)	0.481 (0.045)	0.769 (0.016)	0.534 (0.006)
PCA_10	0.792 (0.100)	0.567 (0.070)	0.799 (0.020)	0.592 (0.002)
PCA_16	0.769 (0.034)	0.628 (0.014)	**0.884** (0.019)	0.639 (0.006)
PCA_22	0.723 (0.043)	0.635 (0.007)	0.786 (0.020)	**0.644** (0.002)
PCA_27	0.754 (0.024)	**0.682** (0.026)	0.770 (0.020)	0.640 (0.007)
All	0.734 (0.066)	0.561 (0.095)	0.705 (0.018)	0.641 (0.003)

Table 6 presents the average precision and recall values from ten models, each trained with a different random seed, per each model configuration. PCA_16 achieves the highest average precision, which is 0.884. Note that users can further adjust the trade-off between precision and recall as needed by shifting the classification threshold of a model.

Answer to RQ3: A classification model that uses multiple similarity functions can significantly outperform single similarity functions. When using the 16 features extracted by PCA, the MLP classifier with the bagging ensemble method achieves the F1 score of 0.743 against the test set, which is significantly higher than the baseline F1 score, 0.479, achieved by the Static-Ronin-TFIDF-Cosine configuration using the learnt threshold.

6 THREATS TO VALIDITY

Threats to internal validity concerns factors that could have affected precise observation and measurement of the effects achieved by our proposed technique. Our heuristic used to extract the ground truth test relationships from the vast logs of bug tickets and test results may not be 100% accurate, given the margin or human error as well as the hyperparameter for the bug ticket creation window (seven days). However, since we ultimately aim to automate and assist the debugging activities of human developers, aiming to match their root cause analysis as closely as possible remains a valuable goal. Further, all bug tickets, including the ones in our test set, have gone through further inspection to identify the bug introducing change. Consequently, we think the test relationship extracted from existing bug tickets are reasonably accurate. We use open-source implementations that withstood public scrutiny whenever possible, e.g., for tokenisation [21] and classification [33].

Threats to external validity concerns factors that may limit the generalisation of our results. With this work, our aim is not to generalise widely, but to specifically assist the development process of SAP HANA2. While all the studied similarity functions are generic (i.e., their design is not specific to SAP HANA2), we cannot expect these functions to perform similarly when applied to projects with completely different histories and contexts. Furthermore, even with SAP HANA2, a fundamental change in test trajectories caused by major version changes or foundational architectural redesigns may

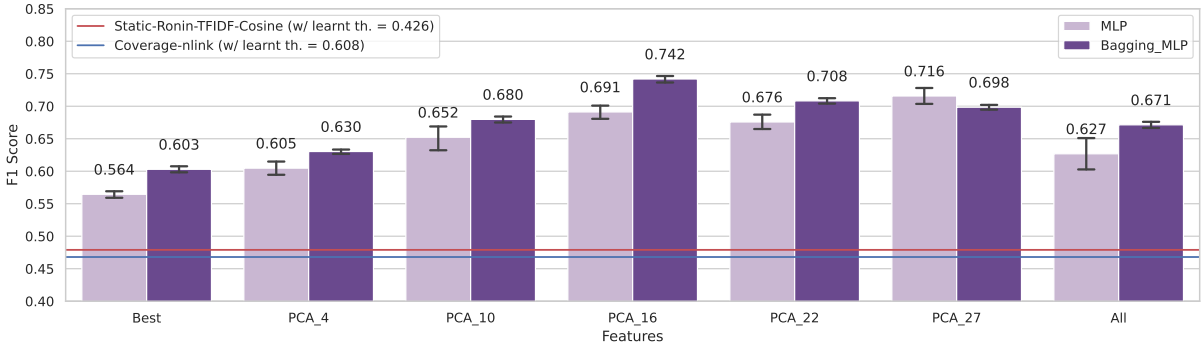

Figure 6: F1 scores of the models using different feature extraction strategies against the test data. Note that the y-axis ranges from 0.40 to 0.85. The red and blue horizontal lines represent the F1 scores for the relevance classifications when using the Static-Ronin-TFIDF-Cosine and Coverage-nlink similarities, respectively, with learnt thresholds. Each error bar indicates the 95% confidence interval.

result in discontinuity in the studied information sources. Only a longitudinal study using additional real world data can provide a more accurate assessment of our proposed approach.

Threats to construct validity arises when the values we observe and report do not actually reflect the properties we aim to measure. All evaluation metrics we use are widely used standard evaluation metrics for classification tasks, minimising this threat.

7 RELATED WORK

Our work is related to several existing approaches in the area of failure clustering. Podgurski et al. [34] showed that certain failures share the same root causes, and clustering those failures facilitates further steps of diagnosing faults. They use coverage profiles as features to perform clustering analysis as well as multivariate visualization of failures. Jones et al. [22] represented a failing test case, (i.e., a test breakage), as a set of program statements whose Tarantula suspiciousness scores [23] are higher than the given threshold; subsequently, failing test cases are clustered using the Jaccard similarity between the sets of high suspiciousness statements. Similarly, Liu et al. [30] ranked program statements according to the likelihood of being faulty, and used the weighted Kendall-Tau distance between rankings to cluster failing test cases. Recently, Gao and Wong proposed MSeer [12], which extends the rank-based approach of Liu et al. with an improved Kendall-tau distance metric.

Golagha et al. [13] proposed a failure clustering method using non-code features such as test history, and evaluated them in the hardware-in-the-loop testing of automotive software. While Golagha et al. simply use the weighted sum of the features they studied to perform agglomerative clustering, we focus on classification of the relevance between two test breakages based on a wider range of features, including test coverage and lexical information from the source code. Moreover, our work directly compares the performance of individual similarity functions as a predictor of shared root causes. We will consider clustering of test breakages based on the classification results as future work.

Our previous work [1] introduced hypergraph-based coverage representation that allows efficient calculation of pairwise similarity of tests. An empirical evaluation on Defects4J [24] showed that hypergraph-based distance, nlink, can produce accurate failure clusterings when used with agglomerative clustering. This paper applies and evaluates the hypergraph-based coverage representation in an industrial context of SAP HANA2.

Test similarity measures have been studied in the context of test case selection and prioritisation, with the aim of improving diversity among test cases. Thomas et al. [38] proposed a static black-box test prioritisation technique that applies topic modelling to the source code of test cases: the similarity between topics are subsequently used to select test cases with distinct functionalities. Historical information has also been shown to be effective when calculating test similarity in rapid release environments for test prioritisation [17]. Chen et al. [6] prioritised a large pool of fuzzer-generated test inputs for compiler testing, by measuring distances between inputs using both coverage profiles and lexical information from the test input (i.e., source code input to the compiler): the aim is to ensure high diversity early in the test execution. Our context differs from existing work in that we perform white-box testing of SAP HANA2 with a focus on identifying shared root causes behind test breakages.

8 CONCLUSION

We propose a technique that can determine whether two test breakages share the same root cause to improve the debugging efficiency in SAP HANA2. Our technique uses a range of information sources, including static, historical, and dynamic information from test cases and their executions. We show that, although using a single similarity function alone does not generalise well to previously unseen data, various similarity functions from different information sources can complement each other. This motivates us to train an unified classification model that uses multiple similarity functions as features. We adopt PCA-based feature extraction and a bagging ensemble method to reduce overfitting. When evaluated using three-month CI data from SAP HANA2, our classification model achieves the F1 score of 0.743, which is at least 55% improvement over a single similarity function. We plan to implement the shared root cause analysis to the CI pipeline of SAP HANA2 to improve the post-submit testing process.

ACKNOWLEDGMENTS

Gabin An, Juyeon Yoon, and Shin Yoo are supported by SAP Labs Korea, as well as by National Research Foundation of Korea (NRF) Grant (NRF-2020R1A2C1013629) and Institute for Information & communications Technology Promotion grant funded by the Korean government (MSIT) (No.2021-0-01001).

REFERENCES

[1] Gabin An, Juyeon Yoon, Joyce Jiyoung Whang, and Shin Yoo. 2021. Improving Test Distance for Failure Clustering with Hypergraph Modelling. *arXiv preprint arXiv:2104.10360* (2021).

[2] Claude Berge. 1984. *Hypergraphs: combinatorics of finite sets*. Vol. 45. Elsevier.

[3] Andrew P Bradley. 1997. The use of the area under the ROC curve in the evaluation of machine learning algorithms. *Pattern recognition* 30, 7 (1997), 1145–1159.

[4] Leo Breiman. 1996. Bagging predictors. *Machine learning* 24, 2 (1996), 123–140.

[5] Simon Butler. 2012. Mining Java class identifier naming conventions. In *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, 1641–1643.

[6] Yang Chen, Alex Groce, Chaoqiang Zhang, Weng-Keen Wong, Xiaoli Fern, Eric Eide, and John Regehr. 2013. Taming compiler fuzzers. In *Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation*. 197–208.

[7] Ermira Daka, José Miguel Rojas, and Gordon Fraser. 2017. Generating Unit Tests with Descriptive Names or: Would You Name Your Children Thing1 and Thing2?. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2017)*. Association for Computing Machinery, New York, NY, USA, 57–67.

[8] Jesse Davis and Mark Goadrich. 2006. The relationship between Precision-Recall and ROC curves. In *Proceedings of the 23rd international conference on Machine learning*. 233–240.

[9] Andrea De Lucia, Massimiliano Di Penta, Rocco Oliveto, Annibale Panichella, and Sebastiano Panichella. 2012. Using IR methods for labeling source code artifacts: Is it worthwhile?. In *2012 20th IEEE International Conference on Program Comprehension (ICPC)*. IEEE, 193–202.

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

[11] Eric Enslen, Emily Hill, Lori Pollock, and K Vijay-Shanker. 2009. Mining source code to automatically split identifiers for software analysis. In *2009 6th IEEE International Working Conference on Mining Software Repositories*. IEEE, 71–80.

[12] Ruizhi Gao and W Eric Wong. 2017. MSeerâĂŤAn advanced technique for locating multiple bugs in parallel. *IEEE Transactions on Software Engineering* 45, 3 (2017), 301–318.

[13] Mojdeh Golagha, Constantin Lehnhoff, Alexander Pretschner, and Hermann Ilmberger. 2019. Failure clustering without coverage. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 134–145.

[14] Mojdeh Golagha, Alexander Pretschner, Dominik Fisch, and Roman Nagy. 2017. Reducing failure analysis time: An industrial evaluation. In *2017 IEEE/ACM 39th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*. IEEE, 293–302.

[15] Samir Gupta, Sana Malik, Lori Pollock, and K Vijay-Shanker. 2013. Part-of-speech tagging of program identifiers for improved text-based software engineering tools. In *2013 21st International Conference on Program Comprehension (ICPC)*. IEEE, 3–12.

[16] Isabelle Guyon and André Elisseeff. 2003. An introduction to variable and feature selection. *Journal of machine learning research* 3, Mar (2003), 1157–1182.

[17] Hadi Hemmati, Zhihan Fang, Mika V Mäntylä, and Bram Adams. 2017. Prioritizing manual test cases in rapid release environments. *Software Testing, Verification and Reliability* 27, 6 (2017), e1609.

[18] Geoffrey E Hinton. 1990. Connectionist learning procedures. In *Machine learning*. Elsevier, 555–610.

[19] Mohammad Hossin and Md Nasir Sulaiman. 2015. A review on evaluation metrics for data classification evaluations. *International journal of data mining & knowledge management process* 5, 2 (2015), 1.

[20] Yanqin Huang, Junhua Wu, Yang Feng, Zhenyu Chen, and Zhihong Zhao. 2013. An empirical study on clustering for isolating bugs in fault localization. In *2013 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*. IEEE, 138–143.

[21] Michael Hucka. 2018. Spiral: splitters for identifiers in source code files. *Journal of Open Source Software* 3, 24 (2018), 653.

[22] James A Jones, James F Bowring, and Mary Jean Harrold. 2007. Debugging in parallel. In *Proceedings of the 2007 international symposium on Software testing and analysis*. 16–26.

[23] James A Jones and Mary Jean Harrold. 2005. Empirical evaluation of the tarantula automatic fault-localization technique. In *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*. 273–282.

[24] René Just, Darioush Jalali, and Michael D Ernst. 2014. 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*. 437–440.

[25] Rafael-Michael Karampatsis, Hlib Babii, Romain Robbes, Charles Sutton, and Andrea Janes. 2020. Big code!= big vocabulary: Open-vocabulary models for source code. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*. IEEE, 1073–1085.

[26] Maurice George Kendall. 1948. Rank correlation methods. (1948).

[27] Vladimir I Levenshtein et al. 1966. Binary codes capable of correcting deletions, insertions, and reversals. In *Soviet physics doklady*, Vol. 10. Soviet Union, 707–710.

[28] Yihan Li and Chao Liu. 2012. Using cluster analysis to identify coincidental correctness in fault localization. In *2012 Fourth International Conference on Computational and Information Sciences*. IEEE, 357–360.

[29] Zachary C Lipton, Charles Elkan, and Balakrishnan Naryanaswamy. 2014. Optimal thresholding of classifiers to maximize F1 measure. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*. Springer, 225–239.

[30] Chao Liu and Jiawei Han. 2006. Failure proximity: a fault localization-based approach. In *Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*. 46–56.

[31] Chao Liu, Xiangyu Zhang, and Jiawei Han. 2008. A systematic study of failure proximity. *IEEE Transactions on Software Engineering* 34, 6 (2008), 826–843.

[32] Qingzhou Luo, Farah Hariri, Lamyaa Eloussi, and Darko Marinov. 2014. An empirical analysis of flaky tests. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 643–653.

[33] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, Jake Vanderplas, Alexandre Passos, David Cournapeau, Matthieu Brucher, Matthieu Perrot, and Édouard Duchesnay. 2011. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research* 12 (2011), 2825–2830.

[34] Andy Podgurski, David Leon, Patrick Francis, Wes Masri, Melinda Minch, Jiayang Sun, and Bin Wang. 2003. Automated support for classifying software failure reports. In *25th International Conference on Software Engineering, 2003. Proceedings.* IEEE, 465–475.

[35] Claude Sammut and Geoffrey I. Webb (Eds.). 2010. *TF–IDF*. Springer US, Boston, MA, 986–987. https://doi.org/10.1007/978-0-387-30164-8_832

[36] Amit Singhal et al. 2001. Modern information retrieval: A brief overview. *IEEE Data Eng. Bull.* 24, 4 (2001), 35–43.

[37] Jeongju Sohn, Gabin An, Jingun Hong, Dongwon Hwang, and Shin Yoo. 2021. Assisting Bug Report Assignment Using Automated Fault Localisation: An Industrial Case Study. In *2021 14th IEEE Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 284–294.

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

[39] Yabin Wang, Ruizhi Gao, Zhenyu Chen, W Eric Wong, and Bin Luo. 2014. WAS: A weighted attribute-based strategy for cluster test selection. *Journal of Systems and Software* 98 (2014), 44–58.

[40] Li Weishi and Xiaoguang Mao. 2014. Alleviating the impact of coincidental correctness on the effectiveness of sfl by clustering test cases. In *2014 Theoretical Aspects of Software Engineering Conference*. IEEE, 66–69.

[41] William E Winkler. 1990. String Comparator Metrics and Enhanced Decision Rules in the Fellegi-Sunter Model of Record Linkage. (1990).

[42] Svante Wold, Kim Esbensen, and Paul Geladi. 1987. Principal component analysis. *Chemometrics and intelligent laboratory systems* 2, 1-3 (1987), 37–52.

[43] Shin Yoo, Mark Harman, Paolo Tonella, and Angelo Susi. 2009. Clustering Test Cases to Achieve Effective and Scalable Prioritisation Incorporating Expert Knowledge. In *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis* (Chicago, IL, USA) *(ISSTA '09)*. Association for Computing Machinery, New York, NY, USA, 201–212.

[44] Dengyong Zhou, Jiayuan Huang, and Bernhard Schölkopf. 2006. Learning with hypergraphs: Clustering, classification, and embedding. *Advances in neural information processing systems* 19 (2006), 1601–1608.