Static Test Case Prioritization Using Pretrained Language Models

Xiaoying Xia
The University of Texas at Austin
Austin, USA
xiaxiaoying@utexas.edu

Yicheng Zhu
The University of Texas at Austin
Austin, USA
yz28285@my.utexas.edu

Abstract

Test Case Prioritization (TCP) refers to prioritizing the test cases that are more important. It is a commonly used technique to reduce the cost of regression testing. Although white-box TCP methods have been well-studied and proved to significantly improve the rate of fault detection, they can be costly in time and resources. We noticed a recently introduced black-box static TCP method, that uses information never used before - linguistic information, and achieved high fault-detection performance. With the rapid development of language models, we wonder if these novel models can further improve the effectiveness and efficiency, and how it performs compared to the traditional white-box methods. In this paper, we proposed a new static black-box TCP method that uses CodeBERT for vectorization and improved its fault-detection performance compared with the original method. We also compare our method with two existing white-box dynamic TCP methods on 5 Java real-world open-source systems. We found our method outperforms the other methods in most cases while staying time-saving in multiple runs.

Keywords: Test Case Prioritization, Topic Models, Pre-Trained Language Models, CodeBERT

ACM Reference Format:
Xiaoying Xia and Yicheng Zhu. 2022. Static Test Case Prioritization Using Pretrained Language Models . In *Proceedings of ACM Conference (Software Testing).* ACM, New York, NY, USA, 6 pages. https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

Test Case Prioritization (TCP) is a widely studied topic in software testing research [6, 9]. The goal of TCP is to reorder the test suite so that the test cases that are easier to fail execute first, which eventually speeds up the testing process.

Software Testing, Dec 2022, Austin, TX, USA

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

TCP is applied in regression testing, as new changes are made to the System Under Test (SUT), we need to ensure that no errors are introduced because of the changes. However, due to the limitation of time and resources, re-executing the whole test suite could be infeasible. From this point, the concept of Regression Test Selection [14], Test Suite Minimization and Test Case Prioritization gained value among researchers. As Regression Test Selection discards test cases, it shows severe loss of fault detection rate [5]. Since TCP does not discard test cases and shows significant improvement in fault detection rate, it is more widely studied in recent years. Researchers have proposed many prioritization criteria that utilize different information in relation to the test cases: Early methods were based on code coverage information [11, 19]. More recently, black-box methods have been proposed. For example, in [16], the researchers focus on the logs of user behavior instead of the source code of the SUT. Most of these TCP methods use the execution information of the SUT, which is the run-time behavior of test cases.

However, it is claimed in [18] that execution information may also be unavailable for the following reasons:

- Collecting execution information can be expensive in time and resources
- For large-scale systems, storing and maintaining the execution information can be costly
- Execution information must be updated frequently updated with the evolution of SUT and test suite

Therefore, to address the situation where execution information is unavailable, researchers have recently developed static TCP techniques. For example, [13] proposed a method based on the static call graph of the test cases. In [12], the researchers treat test cases as string of characters, and proposed a TCP approach that calculates the distance between those strings, to determine the similarity between test cases. In addition, [17] proposed a novel black-box static method that utilized a source of information never used before: *linguistic data*, i.e., the name of identifiers, comments, and string contents to determine the functionality of test cases. To be more specific, this approach uses a text analysis algorithm, called topic modeling, to create topics from linguistic data, and prioritizes test cases with different topics. According to the evaluation process, this method is better than existing black-box static methods as it is more robust from trivial

differences in the source code, and it captures more information than the call graph. In this paper, we will refer to this method as LDA as it models the linguistic data by the LDA algorithm.

With the rapid development in topic modeling algorithms and pre-trained models, we wonder if the improvement of these algorithms can further boost the effectiveness and efficiency of LDA. It is also useful to know how the black-box methods compared to traditional white-box methods in terms of fault-detection rate and time consumed. These points motivated us to propose the following research questions.

- **RQ1** Can we improve the performance of LDA method with advanced topic-modeling methods?
- **RQ2** How do black box methods perform compared with white box ones in terms of effectiveness and efficiency?

For **RQ1**, we used different topic modeling methods and evaluated their fault-detection performance and execution time on two real-world Java systems as used in [17]. We find that our CodeBERT-based technique increases the average fault detection rate over existing techniques by as much as xx%, and increases the execution time by as much xx%. To answer **RQ2**, we choose five different Java benchmark datasets and compare the performance of two existing white-box dynamic methods with three black-box static methods. The evaluation result shows that the proposed method outperforms other methods in terms of fault-detection rate.

In summary, this paper makes the following contributions:

- We introduce a novel black-box static TCP technique, based on a novel language modeling algorithm called CodeBERT.
- We conduct a detailed case study to compare the black-box static methods, and between black-box static and white-box dynamic methods. We find that our proposed technique achieves the same or better performance than existing techniques.

The rest of the paper is organized as follows: Section 2 introduces the methods involved in this project. Section 3 proposes a novel language modeling-based TCP method called CodeBERT4TCP. Section 4 gives the implementation details, involved datasets and evaluation metrics for this project. Section 5 and Section 6 give the evaluation result and conclusion. The code for our work is available on GitHub[1].

2 Methodology

This section introduces the methods involved in this project, including two types of white-box dynamic methods and the original LDA-based TCP method. We select Adaptive Random Test Case Prioritization as the benchmark white-box dynamic method. We also give a short description on the vectorization-based TCP.

2.1 Adaptive Random Test Case Prioritization

In [11], the researchers proposed a series of ART-based TCP techniques guided by dynamic coverage information (ART-D). In each iteration, a candidate set is dynamically created by randomly picking test cases from the set of not-yet-prioritized tests as long as they can increase coverage. The test case within the candidate set that is the furthest away from the set of already-prioritized tests is then selected. The authors proposed and assessed different set distance functions; for this paper, we implemented the version that performed better ("maxmin").

Another ART-based TCP method used in this paper is [19] (ART-F), for which the essence is the same as ART-D. The main differences are in the way the candidate set is created and in the distance metric adopted. While in [11] the candidate set has a flexible size, [19] used a fixed size (i.e., 10) for the candidate set. Besides, the researchers used the Manhattan distance instead of the Jaccard distance in [11]

2.2 LDA based Test Case Prioritization

Algorithm 1 LDA as a generation model

for each document w **do**
  Draw topic distribution $\theta \sim$ Dirichlet (α);
  for each word at position n **do**
    Sample topic $z_n \sim$ Multinomial (θ);
    Sample word $w_n \sim$ Multinomial (β_{z_n});
  end for
end for=0

LDA [1] is a generative probabilistic model for mining the topic distribution of documents. It views the generation of documents as a process of sampling topics and sampling words from topics. The detailed generation process is shown in algorithm 1.

The role LDA played in previous test case prioritization works is vectorization. The vector of each document is its topic vector. The reason why this works is that topic vectors generated by topic models contain high-level topic information of documents and similar documents share similar topic information. Therefore, similar documents would have similar topic vectors and vice versa.

2.3 Vectorization Based TCP

Vectorization-based TCP is first proposed by [17]. It proposed a workflow which followed by many works. Our work also follows this workflow. Vectorization-based TCP contains three steps.

Firstly, preprocess and vectorize the test suite. For each test case, it is vectorized using various models, such as LDA,

[1] https://github.com/Zqjjjydl/Test-case-prioritization-by-text-vectorization

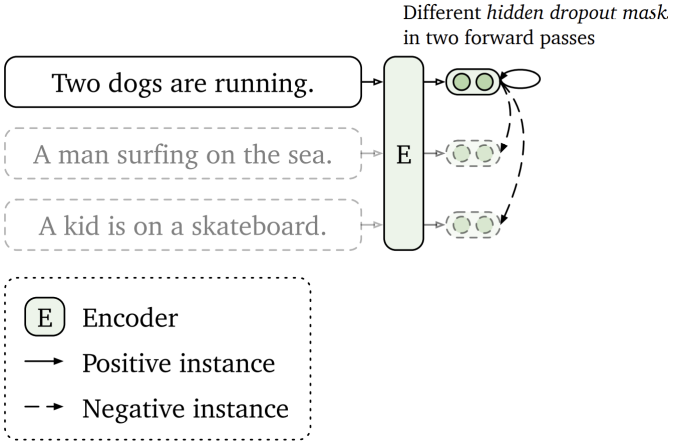
prodLDA, or CodeBERT. What's expected in this step is that the vector of each document contains enough sentiment information of them, which means that sentimentally similar test cases should have similar vectors and vice versa.

Secondly, distances are calculated between each pair of test cases. Several kinds of distance metrics are applied by previous research, including Manhattan distance, Euclidean one, and angular one. In this work, we applied the angular one because our vectors are continuous.

Last but least, calculate the order of test cases. The order is usually calculated using a greedy algorithm. From all test cases that are not yet prioritized, the algorithm selects a vector that is mostly far away from all prioritized test cases. This selection process keeps going on until all test cases have been prioritized. The intuition is that it selects mostly different test cases from all previous ones.

3 CodeBERT4TCP

Figure 1. SimCSE predicts same input sentence with different masks from a mini-batch.



The idea of CodeBERT4TCP is inspired by CodeBERT[7] and SimCSE[8]. What we need is high-quality embedding for java code. The problem with CodeBERT is that CodeBERT outputs embedding for each token instead of embedding the whole sentence. In traditional sentence embedding learning, a large labeled dataset contains labeled pairs of the sentimental related sentence. However, such datasets are not available for code. SimCSE is an unsupervised method that can generate sentence embedding using large-scale of unlabeled data. Therefore, we can generate high-quality code embedding using the large scale of code.

The idea of CodeBERT4TCP is extremely easy. We take a collection of code documents ${x_i}_{i=1}^m$. We construct a set of paired examples $D = (x_i, x_i^+)_{i=1}^m$, where x_i and x_i^+ are sentimental related. We follow the contrastive framework setting in Chen et al. [2] and take a cross-entropy objective with in-batch negatives: let h_i and h_i^+ denote the representation of x_i and x_i^+, the training object for (x_i, x_i^+) with a mini-batch of N pairs is:

$$L_i = -log \frac{e^{sim(h_i, h_i^+)/\tau}}{\sum_{j=1}^{N} e^{sim(h_i, h_j^+)/\tau}} \quad (1)$$

where τ is a temperature hyperparameter and $sim(h_i, h_i^+)$ is the cosine similarity of h_i and h_i^+. In our work, h_i of x_i is computed by CodeBERT.

CodeBERT for embedding generation. CodeBERT is a pre-trained language model trained on the programming language (PL) and natural language (NL). It learns a general representation of code that can support various downstream NL-PL applications such as natural language code search, and code documentation generation. It applied both mask language modeling (MLM) tasks and replaced token detection (RTD) tasks to train a language model.

In the MLM task, a proportion of tokens in original code documents is masked and the language model is trained to predict which word best fits in the masked token. For example, given the sentence "I <mask> eating apple. It tastes good.", the Languege model would fill in the masked place using "love" or "like".

In the RTD task, a proportion of tokens is randomly replaced by other tokens, and the language model is trained to predict whether each token in the input has been replaced. For example, given the sentence "I hate eating apple. It tastes good.", the Languege model would find the words "hate" and "good" to be likely replaced because they can contradictory.

After training on large-scale code documents, CodeBERT gains the general ability to understand code documents and their output embeddings containing plenty of information on code that can be used for TCP. However, we still only have the embeddings for each token in the input document while what we need is embedding for the whole document. Therefore, we need to finetune CodeBERT on pairs of sentimental-related code documents.

Construct sentimental related pairs. The most difficult problem to make CodeBERT able to output meaningful embedding for the whole sentence is to construct sentimental related pairs of code documents. SimCSE proposed a simple but effective method that can work well on unlabeled datasets. It applied dropout as data augmentation on input and the augmented document should be sentimentally similar to the original document.

In standard Bert[3], there are dropout masks. Therefore, we can simply pass the same code document twice into the model and get two embeddings with different drop mask z, $z^{'}$ and our training object would be:

$$L_i = -log \frac{e^{sim(h_i^{z_i}, h_i^{z_i^{'}})/\tau}}{\sum_{j=1}^{N} e^{sim(h_i^{z_i}, h_j^{z_j^{'}})/\tau}} \quad (2)$$

where $h_i^{z_i}$ is the embedding of x_i using dropout mask z_i.

To finetune CodeBERT, we collected 20000 java datapoints from [10] and finetune original CodeBERT using above object following same hyperparameters setting as SimCSE. We only finetuned the last three layers of the original model because of computation resouce limitation. The data comes from publicly available open-source non-fork GitHub repository and are filtered with a set of constraints and rules. For example, each project should be used by at least one other project.

4 Experiment setup

4.1 Evaluation Metrics

In order to evaluate the fault-detection performance, we use the Average Percentage of Faults Detected (APFD) [15]. APFD is calculated according to Equation 3, in which we are given a test suite T containing n test cases and a set F of m faults revealed by T, for each ordering of T we denote as TF_i the position of the first test case that reveals fault i.

$$APFD = 1 - \frac{TF_1 + TF_2 + ... + TF_m}{nm} + \frac{1}{2n} \tag{3}$$

To evaluate the efficiency, we assess the involved TCP methods in terms of execution time, which refers to the actually spent CPU time, which we measured by using Python's time.process_time() function.

4.2 Systems Under Test

4.2.1 SUT for RQ1. In order to verify the improvement of fault-detection performance between CodeBERT4TCP and the original LDA-based method, we choose the same SUT as it. In [17], the researchers choose five systems from the Software-artifact Infrastructure Repository (SIR) [4]. They are all Java-based systems maintained by the Apache Foundation. See table 1 for details of the SUTs. The failure rate column indicates the portion of test cases that have at least one fault. The reason to choose these SUTs is that they are real-world systems that contains enough test cases and faults.

These datasets also provide us with data to evaluate the TCP methods. Specifically, the datasets include a *fault matrix* which indicates which faults each test is able to detect. As the fault matrix is unknown in real-world, it is only used to evaluate the fault-detection rate of each TCP method.

4.2.2 SUT for RQ2. Since SUTs for RQ1 are too outdated to actually run, we can't collect the run-time coverage information from these systerms. Therefore, we selected another five Java systems that are able to execute on the experiment platform. Similar to SUTs for RQ1, SUTs for RQ2 also provide fault matrix to evaluate among TCP methods. According to table 2, the SUTs for RQ2 are generally larger than SUTs for RQ1, so it is easier to compare the efficiency of different methods.

4.3 Experiment Process

To answer RQ1 and RQ2, we applied the investigated TCP approaches to each experimental subject and measured: (i) the execution time; (ii) the APFD of the prioritized test suites. This process was repeated 10 times to account for the random nature of ART-based TCP methods considered in this project. In order to collect coverage information, we used JCov in this project, which is a open-source code coverage collecting tool that produces coverage reports baseed on run-time features of JVM. We use total-statement strategy in [11], as it is reported with good performance.

Other baseline methods are listed as followings:

- Random: Random method is prioritize the test cases in a random order.
- String: Evaluate two test cases' similarity using their edit distance.
- LDA: vectorization using LDA. Preform greedy algorithm on vectors.
- prodLDA: vectorization using ProdLDA. Preform greedy algorithm on vectors.

All the experiments were performed on an 3.1 GHz Intel i5 CPU, 16 GB RAM, running macOS 13.0.1 (22A400).

5 Results

5.1 RQ1: APFD Among Black-Box Static Methods

To evaluate our model among different black-box static methods, we compared it with several baseline methods. The APFD results is shown in table3.

From the table, we can observe that our method outperforms previous methods in most datasets. It also shows comparable performance with other methods in Derby v5 even if it does not perform better than LDA. The reason why our method outperforms baseline methods is because it produces higher quality vectors. Language model trained on large scale of code repositories can produce high quality vectors for new code. Also, the finetuing performed using contrastive learning allows the CodeBERT produce meaningful embeddings for the whole sentence. From table refAPFD$_{C}odeBERT$, $wecanfindthatasthenumberoffinetunedlayersincre$

5.2 RQ2: APFD and efficiency Among Black-Box and White-Box Methods

Table 5 shows the mean APFD result among ART-D, ART-F, LDA, prodLDA ande codeBERT based TCP methods. The method proposed in this paper - codeBERT based method outperforms other methods in 3 of the 5 dataset, and has a comparable performance with the best method in the rest two datasets. We can also observe that the difference of APFD value across datasets is greater than the difference between different TCP methods on the same dataset. This indicates an overlap between both the faults found by black-box and white-box TCP techniques and also the performance of these approaches, as also showed in [9].

Table 1. SUT for RQ1 (from SIR [4])

SUT	Version	# of faults	# of tests	Failure Rate (%)
Ant v7	1.5.3	6	105	17
Derby v1	10.1.2.1	7	98	21
Derby v2	10.1.3.1	9	106	34
Derby v3	10.2.1.6	16	120	22
Derby v5	10.3.1.4	26	53	64

Table 2. SUT for RQ2 (from SIR [4])

SUT	Line of Code	# of faults	# of tests
Closure Compiler	90697	101	221
Commons Lang	21787	39	113
Commons Math	84323	7	385
JfreeChart	96382	26	356
Joda-Time	27801	27	123

Table 3. APFD Results Among Black-Box Static Methods

SUT	RAND	CALLG	STRG	LDA	prodLDA	CodeBERT
Ant v7	70.6	85.5	77.2	84.8	86.6	**88.9**
Derby v1	90.7	91.8	90.4	93.7	94.4	**96.4**
Derby v2	91.5	87.6	90.3	94.6	94.8	**96.7**
Derby v3	92.7	93.4	93.5	94.0	93.6	**98.0**
Derby v5	89.1	73.8	82.1	**96.4**	91.1	95.8

Table 4. APFD Results For Different Versions of CodeBERT

SUT	codeBERT-1	codeBERT-2	codeBERT-3
Ant v1	80.1	87.6	**88.9**
Derby v1	93.3	95.7	**96.4**
Derby v2	92.2	95.2	**96.7**
Derby v3	93.3	93.1	**98.0**
Derby v5	89.3	94.7	**95.8**

In regard to execution time, table 6 gives average execution time for different TCP methods. For white-box methods, since every time the source code or test suite is changed, we need to re-run the prioritization process, we only count the average total execution time. For black-box methods, since we will only need to vectorize those changed test cases, we count the total execution time for the whole test suite (left side) as well as the vectorization time and prioritization time for a single test case. As we can see from the table, while all methods remain in the same level of time complexity, the ART-D is the overall fastest method. However, if we consider the real-world regression testing situation, developers will only need to run the black-box methods once and re-vectorize the updated test cases, which is apparently more efficient in the long-run of software development.

In conclusion, the topic-modeling based black-box static TCP methods have the following advantages:

- Comparable fault-detection performance, better in most cases
- Time-saving, no need to re-vectorize the unchanged test cases
- Require minimum access to the source code and don't need to execute the test suite or SUT
- Provide deteministic ordering after the model is trained

6 Conclusion

In this paper, we first make a systematic review of different categories of TCP methods. Afterwards we propose a novel topic-modeling based TCP method that utilizes CodeBERT to extract linguistic data from test cases and vectorize them to produce prioritization results. Finally, we conduct a empirical case study on 10 large-scale real-world Java applications. The experiment result shows our CodeBERT-based method

Table 5. APFD Results Between Black-Box and White-Box Methods

SUT	ART-D	ART-F	LDA	prodLDA	codeBERT
Closure Compiler	**75.3**	72.1	70.6	73.5	75.2
Commons Lang	91.4	92.6	91.4	91.8	**93.6**
Commons Math	33.5	32.7	38.6	39.1	**39.4**
JfreeChart	50.5	**54.8**	48.8	47.6	50.0
Joda-Time	34.2	32.1	35.7	36.4	**37.1**

Table 6. Execution Time (Second) of Different TCP Methods, for black-box methods, the left part is Vectorization Time for whole test suite and the right part is Vectorization Time for one test case

SUT	ART-D	ART-F	LDA	prodLDA	codeBERT
Closure Compiler	**118.2**	239.6	244.5+0.12	260.4+0.14	255.2+0.12
Commons Lang	**1.5**	4.4	10.4+0.04	12.6+0.04	10.9+0.03
Commons Math	**72.2**	145.0	100.4+0.08	106.8+0.07	120.4+0.08
JfreeChart	**61.8**	65.6	68.8+0.05	70.2+0.05	66.5+0.04
Joda-Time	**13.7**	50.5	26.4+0.02	24.3+0.02	30.1+0.02

exceeds other methods in terms of fault-detection rate. Meanwhile, the execution time of topic-modeling based methods are similar to coverage-based methods in one run, but our method saves time in multiple runs of regression testing.

In the future, we plan to expand our method to applications written in different programming language, which requires a stronger model and a better analysis of the test suite.

References

[1] David M. Blei, Andrew Y. Ng, and Michael I. Jordan. 2003. Latent dirichlet allocation. *Journal of Machine Learning Research* 3 (2003), 993–1022.

[2] Ting Chen, Simon Kornblith, Mohammad Norouzi, and Geoffrey Hinton. 2020. A simple framework for contrastive learning of visual representations. In *International conference on machine learning*. PMLR, 1597–1607.

[3] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805* (2018).

[4] Hyunsook Do, Sebastian Elbaum, and Gregg Rothermel. 2005. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Software Engineering* 10, 4 (2005), 405–435.

[5] S. Elbaum, S. Karre, and G. Rothermel. 2003. Improving web application testing with user session data. In *25th International Conference on Software Engineering, 2003. Proceedings.* 49–59. https://doi.org/10.1109/ICSE.2003.1201187

[6] S. Elbaum, A.G. Malishevsky, and G. Rothermel. 2002. Test case prioritization: a family of empirical studies. *IEEE Transactions on Software Engineering* 28, 2 (2002), 159–182. https://doi.org/10.1109/32.988497

[7] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. 2020. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155* (2020).

[8] Tianyu Gao, Xingcheng Yao, and Danqi Chen. 2021. Simcse: Simple contrastive learning of sentence embeddings. *arXiv preprint arXiv:2104.08821* (2021).

[9] Christopher Henard, Mike Papadakis, Mark Harman, Yue Jia, and Yves Le Traon. 2016. Comparing White-Box and Black-Box Test Prioritization. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*. 523–534. https://doi.org/10.1145/2884781.2884791

[10] Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. 2019. Codesearchnet challenge: Evaluating the state of semantic code search. *arXiv preprint arXiv:1909.09436* (2019).

[11] Bo Jiang, Zhenyu Zhang, Wing Kwong Chan, and TH Tse. 2009. Adaptive random test case prioritization. In *2009 IEEE/ACM International Conference on Automated Software Engineering*. IEEE, 233–244.

[12] Yves Ledru, Alexandre Petrenko, Sergiy Boroday, and Nadine Mandran. 2012. Prioritizing test cases with string distances. *Automated Software Engineering* 19, 1 (2012), 65–95.

[13] Hong Mei, Dan Hao, Lingming Zhang, Lu Zhang, Ji Zhou, and Gregg Rothermel. 2012. A static approach to prioritizing junit test cases. *IEEE transactions on software engineering* 38, 6 (2012), 1258–1275.

[14] Gregg Rothermel and Mary Jean Harrold. 1996. Analyzing regression test selection techniques. *IEEE Transactions on software engineering* 22, 8 (1996), 529–551.

[15] Gregg Rothermel, Roland H Untch, Chengyun Chu, and Mary Jean Harrold. 1999. Test case prioritization: An empirical study. In *Proceedings IEEE International Conference on Software Maintenance-1999 (ICSM'99).'Software Maintenance for Business Change'(Cat. No. 99CB36360)*. IEEE, 179–188.

[16] Sreedevi Sampath, Renee C Bryce, Gokulanand Viswanath, Vani Kandimalla, and A Gunes Koru. 2008. Prioritizing user-session-based test cases for web applications testing. In *2008 1st International Conference on Software Testing, Verification, and Validation*. IEEE, 141–150.

[17] 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.

[18] Lingming Zhang, Ji Zhou, Dan Hao, Lu Zhang, and Hong Mei. 2009. Prioritizing JUnit test cases in absence of coverage information. In *2009 IEEE International Conference on Software Maintenance*. IEEE, 19–28.

[19] Zhi Quan Zhou, Arnaldo Sinaga, and Willy Susilo. 2012. On the fault-detection capabilities of adaptive random test case prioritization: Case studies with large test suites. In *2012 45th Hawaii International Conference on System Sciences*. IEEE, 5584–5593.