Optimizing Test Prioritization via Test Distribution Analysis*

Junjie Chen
Yiling Lou
HCST (Peking University), MoE
China
{chenjunjie,louyiling}@pku.edu.cn

Lingming Zhang
Department of Computer Science,
University of Texas at Dallas
USA
lingming.zhang@utdallas.edu

Jianyi Zhou
HCST (Peking University), MoE
China
zhoujianyi@pku.edu.cn

Xiaoleng Wang
Baidu Online Network Technology
(Beijing) Co., Ltd
China
wangxiaoleng@baidu.com

Dan Hao†
Lu Zhang
HCST (Peking University), MoE
China
{haodan,zhanglucs}@pku.edu.cn

ABSTRACT

Test prioritization aims to detect regression faults faster via reordering test executions, and a large number of test prioritization techniques have been proposed accordingly. However, test prioritization effectiveness is usually measured in terms of the average percentage of faults detected concerned with the *number of test executions*, rather than the *actual regression testing time*, making it unclear which technique is optimal in *actual regression testing time*. To answer this question, this paper first conducts an empirical study to investigate the actual regression testing time of various prioritization techniques. The results reveal a number of practical guidelines. In particular, no prioritization technique can always perform optimal in practice.

To achieve the optimal prioritization effectiveness for any given project in practice, based on the findings of this study, we design learning-based Predictive Test Prioritization (PTP). PTP predicts the optimal prioritization technique for a given project based on the test distribution analysis (i.e., the distribution of test coverage, testing time, and coverage per unit time). The results show that PTP correctly predicts the optimal prioritization technique for 46 out of 50 open-source projects from GitHub, outperforming state-of-the-art techniques significantly in regression testing time, e.g., 43.16% to 94.92% improvement in detecting the first regression fault. Furthermore, PTP has been successfully integrated into the practical testing infrastructure of Baidu (a search service provider with over 600M monthly active users), and received positive feedbacks from the testing team of this company, e.g., saving beyond 2X testing costs with negligible overheads.

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**;

KEYWORDS

Regression Testing, Test Prioritization, Machine Learning

ACM Reference Format:
Junjie Chen, Yiling Lou, Lingming Zhang, Jianyi Zhou, Xiaoleng Wang, Dan Hao, and Lu Zhang. 2018. Optimizing Test Prioritization via Test Distribution Analysis. In *Proceedings of the 26th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '18), November 4–9, 2018, Lake Buena Vista, FL, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3236024.3236053

1 INTRODUCTION

Test prioritization [23, 60, 64, 69, 75], one of the most widely studied regression testing approaches [37, 40, 66, 78], aims to reorder test executions with the goal of detecting more faults earlier [67]. To date, based on whether considering test execution costs, various *cost-unaware* techniques [55, 74] and *cost-aware* techniques [35, 61] have been proposed. These existing prioritization techniques are mostly evaluated based on Average Percentage of Faults Detected (APFD) [44, 58], which calculates the average rate of faults detected when executing different numbers of tests. However, APFD assumes that all the tests have the same execution time and treats them equivalently [33], which is usually not true in practice. For example, for the project MapDB [7] used in this paper, the test with the longest running time spends $8.8*10^5$X more time than that with the shortest running time. To address this measurement issue, Elbaum et al. [33] proposed a cost-cognizant version of APFD — $APFD_c$, which considers different test costs and fault severities. Since fault severities can be hard to determine in practice, Epitropakis et al. [35] further simplified this measurement by assuming all faults have the same severity. However, to date, there lack extensive studies uniformly comparing both cost-unaware and cost-aware techniques in terms of both APFD and $APFD_c$, and thus it is yet unknown which test prioritization technique performs optimal in practice.

To answer this question, we first conduct an empirical study of widely-used prioritization techniques in terms of both APFD and

*This work is partially supported by National Key Research and Development Program of China Grant No. 2017YFB1001803, NSFC Grant Nos. 61672047, 61529201, 61522201, and 61861130363; it is also partially supported by NSF Grant Nos. CCF-1566589, CCF-1763906, UT Dallas start-up fund, Google, Huawei, and Samsung.
†Corresponding author
‡HCST is short for Key Lab of High Confidence Software Technologies.

ESEC/FSE '18, November 4–9, 2018, Lake Buena Vista, FL, USA

ACM ISBN 978-1-4503-5573-5/18/11…$15.00
https://doi.org/10.1145/3236024.3236053

$APFD_c$ on 14 open-source projects from GitHub. The studied techniques include four widely-studied cost-unaware techniques (i.e., the total technique, the additional technique [67], the search-based technique [55], and the adaptive random technique [50]), a state-of-the-art cost-aware technique (i.e., the cost-cognizant additional technique [61]), and also a baseline technique (i.e., the *cost-only* technique that simply prioritizes tests based on the ascending order of test execution time). Our study reveals various practical guidelines. In particular, there is no type of test prioritization techniques that can always perform optimal in terms of the $APFD_c$ measurement. For example, the state-of-the-art additional technique does not perform well in terms of $APFD_c$, while the minimalist cost-only technique can perform rather well in some cases.

To achieve the optimal prioritization effectiveness for any given project in practice, we further propose a Predictive Test Prioritization approach (PTP), which optimizes test prioritization by guiding the selection of prioritization techniques (i.e., cost-unaware, cost-aware, and cost-only techniques) via machine learning. There are two main challenges in PTP. The first one is which characteristics can impact the selection of prioritization techniques. Through quantitatively and qualitatively analyzing the study results, we find that the distribution of test coverage, testing time, and coverage per unit time can greatly influence the effectiveness of prioritization techniques. This is also the key insight of PTP. The test coverage, testing time, and coverage per unit time of each test are regarded as basic features, and then how to utilize such basic features to predict the optimal prioritization technique is the second challenge. Based on the observations in the study, the high-level distribution patterns constructed from these basic features are more directly related to the optimal prioritization. Therefore, in PTP we adopt the recently proposed XGBoost (short for eXtreme Gradient Boosting) algorithm that can effectively learn high-level features [28].

PTP builds a predictive model via XGBoost by collecting three groups of features (i.e., distribution of test coverage, testing time, and coverage per time unit) on existing projects and labeling which prioritization technique performs optimal on the training data. Based on this model, for a new project PTP can directly predict which prioritization technique performs optimal in practice. To sufficiently evaluate the effectiveness of PTP, we conduct a study on the same 14 projects and additional 36 open-source projects from GitHub (50 projects in total). From the results, PTP correctly predicts the optimal technique for 46 (out of 50) projects, indicating extremely high accuracy of 92%. In fact, PTP can also be viewed as a new prioritization technique, which outperforms the existing cost-unaware, cost-aware, and cost-only techniques to a large extent, e.g., 43.16% to 94.92% faster in detecting the first regression fault.

In addition, PTP has been integrated into the practical testing infrastructure of Baidu, a famous search service provider with over 600M monthly active users. In this paper, we report all the results of PTP on the projects of Baidu collected until Feb. 2018, including 11 industrial subjects with 499 real regression faults. The results show that PTP correctly predicts the optimal prioritization technique for 9 of 11 subjects, a high accuracy of 81.82%. Same to the findings on open-source projects, PTP also outperforms the existing cost-unaware, cost-aware, and cost-only techniques significantly, e.g., 16.18% to 83.79% faster in detecting the first real regression fault, further demonstrating PTP's effectiveness in practice. In this industry application, we use the model trained based on the data of the 50 open-source projects to predict the optimal prioritization technique for industrial subjects directly, indicating that PTP provides a generally usable model for its practical usage. In particular, PTP received positive feedbacks from the testing team of this company, e.g., *saving beyond 2X testing costs with negligible overheads.*

This paper investigates which prioritization technique should be applied to a given project in practice through an empirical study and a learning-based approach, and has the following contributions:

- **Empirical Study.** An empirical study comparing various test prioritization techniques in terms of actual regression testing time, which provides a series of practical guidelines to advance test prioritization.
- **Practical Approach.** A machine-learning based approach that predicts which test prioritization technique is optimal in practice for a given project based on its distribution of test coverage, testing time, and coverage per unit time.
- **Effectiveness Evaluation.** Experimental results on 50 open-source projects demonstrating that the proposed approach outperforms state-of-the-art test prioritization techniques by up to 94.92% in detecting the first regression fault.
- **Industry Application.** Integration of PTP in the industrial testing infrastructure, achieving up to 83.79% faster detection of the first regression fault compared with state-of-the-art techniques on 11 industrial subjects with 499 real regression faults.

2 STUDY ON OPTIMAL TEST PRIORITIZATION

To learn which technique performs optimal in terms of practical effectiveness, we perform an empirical study in this section.

2.1 Study Design

2.1.1 Subjects. In this work, we used 50 open-source projects from GitHub, totaling 1,548,339 lines of source code and 8,879.295 seconds of testing time. Note that we used 14 projects in this study, and used all the projects in the study evaluating our learning-based approach PTP (presented in Section 3). For the 14 projects used in this study, 7 projects are randomly selected from prior prioritization work [58, 59]. Besides, we collected another 7 open-source projects from GitHub, each of which has at least 7 minutes testing time, including Camel-core [1], Chukwa [2], Commons-Pool [4], HBase [6], MapDB [7], OpenTripPlanner [8], and PHP [9], which have been used in other testing/debugging topics [27, 39, 52, 53, 79]. Table 1 presents the basic information of all the 50 subjects. As the execution time of a test may vary slightly in different executions, we ran each test 10 times and used the average execution time. Our study is conducted on a workstation with eight-core Intel Xeon E5620 CPU(2.4GHz), 24G memory, and Ubuntu 12.04.5.

2.1.2 Faults. As the existing studies [14, 51] have demonstrated mutation faults to be suitable for software testing experimentation, same as prior work [24–26, 32, 80], we used mutation faults in our study since it is quite challenging to find a large number of real regression faults [58]. Following prior work [58, 59], for each subject we first generated all mutants (i.e., mutation faults) and randomly selected 500 mutation faults, each of which can be detected by at least one test. Then, we constructed 100 mutation groups each of

Table 1: Open-source subjects from GitHub

ID	Subjects	SLOC	TLOC	#Test	Time (s)
1	assertj-core	13,361	53,059	2,470	65.378
2	asterisk-java	30,495	4,263	217	15.078
3	Camel-Core	120,248	134,036	5,630	1,562.098
4	Chukwa	32,654	8,051	131	1,042.183
5	Commons-Pool	5,206	8,232	272	421.401
6	gson-fire	895	726	36	4.023
7	HBase	66,630	17,385	756	1,474.309
8	jasmine-maven-plugin	1,671	1,931	102	54.156
9	java-apns	1,503	1,724	87	2.253
10	LastCalc	4,522	581	32	14.332
11	MapDB	11,871	36,368	4,132	1,048.109
12	OpenTripPlanner	64,718	14,207	379	637.012
13	PHP	755,522	21,983	9,691	902.275
14	vraptor-archive	16,910	16,213	1,130	79.798
15	blueflood	19,517	15,774	961	147.577
16	cloudhopper-smpp	7,081	5,254	193	6.390
17	commons-dbcp	11,592	8,752	560	83.249
18	commons-email	2,734	3,849	174	5.792
19	commons-math	86,748	90,798	5,082	123.219
20	commons-io	9,980	19,189	1,081	137.940
21	cucumber-reporting	2,245	1,255	124	2.641
22	ddd-cqrs-sample	3,494	1,008	24	2.485
23	dictomaton	2,997	1,102	53	11.969
24	DotCi	10,038	2,714	190	18.232
25	ews-java-api	45,313	1,328	99	4.700
26	exp4j	1,086	3,531	284	8.959
27	gdx-artemis	1,851	1,492	35	2.465
28	geo	764	931	92	6.769
29	geohash-java	920	928	56	5.283
30	hivemall	28,569	3,975	150	222.805
31	HTTP-Proxy-Servlet	493	410	24	2.686
32	jackson-core	18,715	9,880	346	10.724
33	jackson-datatype-guava	2,217	1,035	73	5.887
34	jopt-simple	1,924	5,903	727	5.110
35	joss	7,972	6,029	531	16.061
36	jsprit	23,073	18,373	1,250	106.348
37	la4j	7,086	4,050	625	19.019
38	lambdaj	3,634	4,914	265	3.562
39	languagetool	47,589	20,778	719	264.476
40	metrics-core	2,835	2,194	150	2.041
41	raml-java-parser	8,696	3,005	198	4.540
42	redline-smalltalk	5,648	480	43	4.356
43	RoaringBitmap	17,807	21,494	1,148	171.098
44	rome	11,647	2,705	475	7.422
45	scribe-java	2,808	2,536	99	1.121
46	spring-data-solr	8,252	9,329	636	8.441
47	spring-retry	2,729	3,410	186	6.651
48	stream-lib	4,835	3,806	141	116.094
49	tamper	4,330	2,768	62	2.704
50	webbit	4,914	2,442	131	8.074
	Total	1,548,339	606,180	42,052	8,879.295

* Columns 3-6 present the lines of source code, lines of test code, number of tests, and the testing time (in seconds) (i.e., the time spent on executing the whole test suite).

which contains 5 randomly selected mutation faults. That is, we created 100 faulty versions for each subject (each version contains 5 mutation faults). If the total number of mutation faults is less than 500, the number of mutant groups is less than 100. Following prior work [71, 83], we used PIT [10, 29] and MutGen [15] to generate mutation faults for Java and C projects respectively.

2.1.3 Studied Test Prioritization Techniques. Considering the exclusion/inclusion of testing costs, we classify the existing prioritization techniques into two types, cost-unaware and cost-aware techniques. Besides, we implemented a cost-only technique, which prioritizes tests only based on their costs. That is, we implemented three types of prioritization techniques (abbreviated as **Unaware**, **Aware**, and **Only** in the figures and tables) in total. As the mostly studied testing costs are time spent on running each test (abbreviated as testing time) [35, 61], in this paper we use testing time to represent testing costs.

•**Cost-unaware Test Prioritization** refers to the prioritization techniques without balancing testing time and other factors (e.g., test coverage) in prioritization, including the following techniques.

Total and Additional Prioritization are both greedy algorithms [55, 80]. The total technique prioritizes tests based on the descending order of the number of program elements (e.g., statements) covered by the tests, whereas the additional technique prioritizes tests based on the number of program elements that are uncovered by already selected tests but covered by these unselected tests. Although conceptually simple, the additional technique has been widely recognized as a state-of-the-art technique [55, 58, 59, 80].

Search-based Prioritization regards all the permutations of a test suite as candidate solutions and uses some heuristics to guide the process of searching for a better execution ordering of tests. As the genetic algorithm is evaluated to be effective in test prioritization [55], we use it as the representative in search-based prioritization. It first randomly generates a set of permutations, and then forms the next generation through *crossover* and *mutation* operations. For crossover operation, each pair of permutations in the population is regarded as parent permutations to generate two offspring permutations through crossover on a random position. For mutation operation, it randomly selects two tests and exchanges their positions for each offspring permutation. Same as prior work [58], for the genetic algorithm we set the size of population to be 100, the number of generations to be 300, and the probabilities for crossover and mutation to be 0.8 and 0.1.

Adaptive Random Prioritization is proposed to prioritize tests based on their diversity [50]. It defines a *test set distance* to determine which test is selected next during prioritization. Here we use the *test set distance* that is defined to select a test that has the largest *minimum* distance with the already selected tests as the representative, since it is evaluated to be more effective and efficient among all the proposed *test set distances* [50].

We abbreviate the four cost-unaware techniques, the total, additional, search-based, and adaptive random prioritization, as **Tot**, **Add**, **Sea**, and **ARP** in the figures and tables.

•**Cost-aware Test Prioritization** balances testing time and other factors in test prioritization. In this study, we use cost-cognizant additional prioritization [35, 61], which is a typical cost-aware technique. It is firstly proposed to leverage testing time and fault severities in test prioritization [61]. However, as fault severities are rarely available in practice, Epitropakis et al. [35] adapted this technique by ignoring fault severities. Although Epitropakis et al. [35] proposed another cost-aware technique, in our paper we used only the adapted cost-cognizant additional technique because another technique requires the fault history, which is hard to collect in practice, and both techniques are shown to represent state-of-the-art cost-aware prioritization in terms of $APFD_c$ [35]. More specifically, this (adapted) cost-cognizant additional technique prioritizes tests based on the number of program elements uncovered by already selected tests but covered by these unselected tests, *per unit time*.

•**Cost-only Test Prioritization** schedules the execution order of tests based on testing time alone, ignoring other factors (including test coverage). More specifically, the cost-only technique prioritizes tests based on the ascending order of the execution time of each test. In this paper, this technique is treated as the baseline.

Table 2: Test prioritization comparison in terms of APFD

ID	Statement						Method					
	Unaware				Aware	Only	Unaware				Aware	Only
	Tot	Add	Sea	ARP			Tot	Add	Sea	ARP		
1	0.791	0.850	0.728	0.763	0.839	0.447	0.755	0.809	0.732	0.739	0.803	0.447
2	0.587	0.862	0.832	0.734	0.780	0.519	0.679	0.804	0.782	0.743	0.741	0.519
3	0.788	0.951	0.869	0.874	0.905	0.614	0.788	0.931	0.874	0.869	0.900	0.614
4	0.574	0.750	0.717	0.685	0.701	0.521	0.545	0.695	0.712	0.670	0.675	0.521
5	0.592	0.823	0.795	0.748	0.747	0.536	0.595	0.808	0.798	0.743	0.767	0.536
6	0.751	0.838	0.833	0.769	0.793	0.492	0.742	0.772	0.781	0.745	0.724	0.492
7	0.705	0.878	0.795	0.707	0.767	0.541	0.687	0.868	0.766	0.701	0.765	0.541
8	0.683	0.812	0.805	0.748	0.813	0.578	0.656	0.782	0.776	0.726	0.780	0.578
9	0.716	0.853	0.855	0.776	0.793	0.651	0.696	0.820	0.809	0.780	0.776	0.651
10	0.782	0.800	0.795	0.744	0.674	0.559	0.757	0.765	0.740	0.744	0.648	0.559
11	0.691	0.953	0.843	0.871	0.894	0.686	0.667	0.880	0.820	0.887	0.849	0.686
12	0.886	0.956	0.928	0.828	0.773	0.394	0.880	0.952	0.933	0.815	0.790	0.394
13	0.254	0.910	0.431	0.957	0.704	0.147	0.251	0.838	0.478	0.946	0.702	0.147
14	0.698	0.853	0.819	0.765	0.838	0.555	0.711	0.848	0.806	0.773	0.834	0.555

* Each row presents the average APFD values of all mutation groups for each subject and the cells marked with the shading represent the highest APFD values among all the studied techniques on each subject.

2.1.4 Independent and Dependent Variables. In our study, we used two widely-used coverage criteria (i.e. statement coverage and method coverage) in test prioritization. We used Clover [3] and Gcov [5] to collect test coverage (i.e., statement/method coverage) for Java and C projects respectively. To measure the effectiveness of prioritization techniques, we used both APFD and APFD$_c$. The formula ($APFD = 1 - \frac{TF_1+TF_2+\ldots+TF_m}{n*m} + \frac{1}{2n}$) presents the APFD value calculation for a project with n tests and m faults. Note that TF_i is the ranking of the first test in the prioritized test suite that detects the ith fault. APFD$_c$ is extended from APFD by considering both fault severities and test execution cost [33]. As it is hard to acquire the fault severities for each project in practice, following the prior work [35], we simplify APFD$_c$ by regarding all the faults have the same severity (shown in $APFDc = \frac{\sum_{i=1}^{m}(\sum_{j=TF_i}^{n} t_j - \frac{1}{2}t_{TF_i})}{\sum_{j=1}^{n} t_j * m}$, where t_j represents the execution time of the jth test).

2.2 Results and Analysis

2.2.1 Quantitative Analysis. In this section, we quantitatively evaluated the studied test prioritization techniques in terms of both APFD and APFD$_c$.

Table 3: ANOVA analysis and Tukey's HSD test of test prioritization techniques in terms of APFD

ID	Statement						Method					
	Unaware				Aware	Only	Unaware				Aware	Only
	Tot	Add	Sea	ARP			Tot	Add	Sea	ARP		
1	*b*	*a*	*c*	*bc*	*a*	*d*	*b*	*a*	*b*	*b*	*a*	*c*
2	*d*	*a*	*a*	*c*	*b*	*e*	*c*	*a*	*ab*	*b*	*b*	*d*
3	*c*	*a*	*b*	*b*	*b*	*d*	*c*	*a*	*b*	*b*	*ab*	*d*
4	*c*	*a*	*ab*	*b*	*b*	*d*	*b*	*a*	*a*	*a*	*a*	*b*
5	*c*	*a*	*a*	*b*	*b*	*d*	*c*	*a*	*a*	*b*	*ab*	*d*
6	*c*	*a*	*a*	*bc*	*b*	*d*	*ab*	*a*	*a*	*ab*	*b*	*c*
7	*c*	*a*	*b*	*c*	*b*	*d*	*c*	*a*	*b*	*c*	*b*	*d*
8	*c*	*a*	*a*	*b*	*a*	*d*	*c*	*a*	*a*	*b*	*a*	*d*
9	*c*	*a*	*a*	*b*	*b*	*d*	*b*	*a*	*a*	*a*	*a*	*b*
10	*ab*	*a*	*a*	*b*	*c*	*d*	*a*	*a*	*a*	*a*	*b*	*c*
11	*d*	*a*	*c*	*bc*	*b*	*d*	*c*	*a*	*b*	*a*	*ab*	*c*
12	*b*	*a*	*a*	*c*	*d*	*e*	*b*	*a*	*a*	*c*	*c*	*d*
13	*e*	*b*	*d*	*a*	*c*	*f*	*e*	*b*	*d*	*a*	*c*	*f*
14	*c*	*a*	*a*	*b*	*a*	*d*	*c*	*a*	*ab*	*b*	*a*	*d*

* *a* means the group with significantly best effectiveness.

APFD Results. Table 2 shows that the comparison results of prioritization techniques in terms of APFD. From this table, cost-unaware prioritization performs the best in terms of APFD for almost all of subjects and both coverage criteria. Moreover, the additional technique outperforms the other three cost-unaware techniques in most cases, which is consistent with prior studies [55, 58].

To investigate whether these techniques have significant differences in terms of APFD, we performed *one-way ANOVA analysis* [76] at the significance level of 0.05. In addition, we performed *Tukey HSD post-hoc test* to rank the effectiveness of these techniques as different groups. Table 3 shows the analysis results. All p-values in the one-way ANOVA analysis are much less than 0.05, which means that these techniques have significant differences on each subject. On the whole, at least one cost-unaware technique significantly performs better than cost-aware prioritization in most cases. Moreover, cost-unaware prioritization always significantly outperforms cost-only prioritization. That is, cost-unaware test prioritization is the most effective in terms of APFD. Among the four cost-unaware techniques, consistent with prior work [54, 58, 59, 80], the total technique performs the worst in most cases, while the additional technique performs the best in most cases.

Table 4: Test prioritization comparison in terms of APFDc

ID	Statement						Method					
	Unaware				Aware	Only	Unaware				Aware	Only
	Tot	Add	Sea	ARP			Tot	Add	Sea	ARP		
1	0.786	0.842	0.724	0.760	0.837	0.472	0.753	0.802	0.728	0.736	0.800	0.472
2	0.318	0.738	0.686	0.771	0.971	0.981	0.136	0.648	0.585	0.811	0.878	0.981
3	0.667	0.922	0.859	0.890	0.971	0.961	0.668	0.906	0.873	0.880	0.956	0.961
4	0.272	0.594	0.553	0.744	0.909	0.908	0.257	0.521	0.613	0.580	0.865	0.908
5	0.330	0.532	0.473	0.807	0.904	0.966	0.368	0.752	0.853	0.809	0.888	0.966
6	0.540	0.550	0.502	0.579	0.928	0.903	0.529	0.532	0.458	0.561	0.806	0.903
7	0.493	0.826	0.665	0.710	0.966	0.960	0.504	0.801	0.752	0.757	0.960	0.960
8	0.677	0.799	0.802	0.737	0.818	0.606	0.650	0.767	0.768	0.714	0.781	0.606
9	0.293	0.576	0.598	0.830	0.966	0.936	0.273	0.550	0.538	0.803	0.953	0.936
10	0.835	0.880	0.886	0.696	0.995	0.995	0.837	0.969	0.881	0.525	0.989	0.995
11	0.315	0.428	0.840	0.732	0.805	0.981	0.374	0.487	0.706	0.609	0.833	0.981
12	0.624	0.920	0.888	0.873	0.973	0.977	0.619	0.919	0.883	0.788	0.977	0.977
13	0.406	0.897	0.418	0.920	0.778	0.343	0.379	0.771	0.479	0.875	0.695	0.343
14	0.812	0.860	0.919	0.584	0.953	0.897	0.817	0.909	0.895	0.704	0.954	0.897

Table 5: ANOVA analysis and Tukey's HSD test of test prioritization techniques in terms of APFDc

ID	Statement						Method					
	Unaware				Aware	Only	Unaware				Aware	Only
	Tot	Add	Sea	ARP			Tot	Add	Sea	ARP		
1	*b*	*a*	*c*	*bc*	*a*	*d*	*b*	*a*	*b*	*b*	*a*	*c*
2	*d*	*b*	*c*	*b*	*a*	*a*	*f*	*d*	*e*	*c*	*b*	*a*
3	*d*	*b*	*c*	*bc*	*a*	*a*	*c*	*b*	*b*	*b*	*a*	*a*
4	*d*	*c*	*c*	*b*	*a*	*a*	*d*	*c*	*b*	*b*	*a*	*a*
5	*f*	*d*	*e*	*c*	*b*	*a*	*e*	*d*	*bc*	*c*	*b*	*a*
6	*bc*	*bc*	*c*	*b*	*a*	*a*	*c*	*c*	*d*	*c*	*b*	*a*
7	*e*	*b*	*d*	*c*	*a*	*a*	*d*	*b*	*c*	*c*	*a*	*a*
8	*c*	*a*	*a*	*b*	*a*	*d*	*c*	*a*	*a*	*b*	*a*	*c*
9	*d*	*c*	*c*	*b*	*a*	*a*	*d*	*c*	*c*	*b*	*a*	*a*
10	*c*	*bc*	*b*	*d*	*a*	*a*	*b*	*a*	*b*	*c*	*a*	*a*
11	*e*	*d*	*b*	*c*	*b*	*a*	*f*	*e*	*c*	*d*	*b*	*a*
12	*d*	*b*	*bc*	*c*	*a*	*a*	*e*	*b*	*c*	*d*	*a*	*a*
13	*c*	*a*	*c*	*a*	*b*	*d*	*e*	*b*	*d*	*a*	*c*	*f*
14	*d*	*c*	*ab*	*e*	*a*	*bc*	*c*	*b*	*b*	*d*	*a*	*b*

APFD$_c$ Results. We compared these techniques in terms of APFD$_c$, whose results are shown in Table 4. The distribution of shadings in this table is totally different from that in Table 2. From Table 4, each of the three types of test prioritization can perform the best in some cases, even the minimalist cost-only technique performs rather well on some subjects while other techniques do not perform well. For example, when using method coverage, cost-unaware, cost-aware, and cost-only test prioritization performs the best on 2/5/7 subjects respectively. Among the four cost-unaware techniques, the additional technique does not have obvious advantages compared with the others, but the ARP technique is competitive with the additional technique in terms of APFD$_c$. We suspect the reason to be that all the techniques except ARP tend to execute earlier the tests with higher coverage (which usually cost more time), leading to delayed fault detection in terms of APFD$_c$.

To confirm our findings, we also performed *one-way ANOVA analysis* and *Tukey HSD post-hoc test*, whose results are shown in Table 5. All p-values are much less than 0.05, indicating these techniques also have significant differences in terms of APFD$_c$ on each subject. However, different from the APFD results in Table 3, there is not any type of test prioritization that clearly outperforms the others on the whole. Moreover, among the four cost-unaware techniques, no technique dominantly performs better than the others.

Overall, we get the following findings from this study:

> The cost-cognizant $APFD_c$ measurement has different trends compared with the widely-used APFD, indicating that $APFD_c$ instead of APFD should be used in test prioritization research.

> No type of test prioritization always performs better than the others in terms of $APFD_c$. Surprisingly, even the cost-only technique can outperform other techniques in some cases.

2.2.2 Qualitative Analysis. In this section, we qualitatively analyzed in-depth reasons for prioritization effectiveness. Since different types of techniques tend to have different performance for different subjects, we suspect one major difference among the three types of techniques, the prioritization criteria (i.e., test coverage, testing time, and coverage per unit time), may impact prioritization effectiveness. Therefore, we analyzed the distribution of test coverage, testing time, and coverage per unit time. Here we analyzed the test distribution for six representative subjects because the others have similar conclusions with at least one of these subjects, shown by Figure 1. We used statement coverage as the representative. The x-axis represents the percentage of tests, where we rank the tests based on the ascending order of the values of testing time, test coverage, and coverage per unit time, respectively. The y-axis represents the logarithm of testing time, test coverage, or coverage per unit time. We deal with the exponent by transforming them to be larger than one to make the logarithm values always positive.

Based on Table 4, the cost-only technique performs optimal for *Commons-Pool* (ID is 5) and *LastCalc* (ID is 10), the cost-unaware technique performs optimal for *assertj-core* (ID is 1), and *PHP* (ID is 13), and the cost-aware technique performs optimal for *gson-fire* (ID is 6) and *java-apns* (ID is 9). From Figure 1, the subjects with the same conclusions tend to have similar distribution of test coverage, testing time, and coverage per unit time, which confirms our hypothesis. For example, for *assertj-core* and *PHP*, the testing time of most tests is quite close and the testing time of only a small number of tests is largely different from others, and thus the cost-only technique performs the worst. For *Commons-Pool* and *LastCalc*, the distribution of testing time is more uneven than that of test coverage and coverage per unit time, and thus the advantage of the cost-only technique becomes more distinct. That is, these distribution patterns make some technique perform better or worse.

> The distribution of test coverage, testing time, and coverage per unit time has important impacts on the determination of which prioritization technique (the cost-unaware, cost-aware, and cost-only techniques) should be applied to a specific project in practice.

3 PREDICTIVE TEST PRIORITIZATION

Based on the above findings, projects with similar distribution of test coverage, testing time, and coverage per unit time tend to have the same optimal prioritization technique. That is, based on the distribution of existing projects, it is possible to build a predictive model to predict the optimal prioritization technique for a new project in order to achieve the fastest fault detection in practice. Based on this insight, we propose the first **P**redictive **T**est **P**rioritization approach, abbreviated as PTP, which predicts the optimal prioritization technique for a specific project in advance via machine learning (described in Section 3.1), and then evaluate the effectiveness of PTP (described in Section 3.2 and Section 3.3).

3.1 PTP Approach

Figure 2 shows the overview of PTP. In general, PTP builds a predictive model to predict the optimal prioritization technique for any given project in practice based on various test distribution features, such as the distribution of test coverage, testing time, and coverage per unit time. In the training process, PTP collects the test distribution features and label information (e.g., which prioritization technique performs optimal) for each training project, and performs feature normalization and over-sampling to build the predictive model. Then, given any new project, PTP can predict its optimal test prioritization technique based on its test distribution features. Note that it is impossible to collect the test distribution features without running the project. Therefore, following all the existing work in test prioritization [59, 80], PTP uses the test distribution information of the previous version instead. That is, PTP applies the predictive model on the specific project by using the distribution information of a previous version. We next describe the details for building the PTP predictive model:

3.1.1 Feature Collection. Since the qualitative analysis in Section 2.2.2 shows that the optimal prioritization technique for a specific project is related to its distribution of test coverage, testing time, and coverage per unit time, PTP regards these information as the features of a predictive model. That is, for each project with a test suite, we extract the three group of features: (1) the number of program elements covered by each test, (2) the testing time of each test, and (3) the number of program elements covered by each test per unit time. Although coverage per unit time can be calculated based on test coverage and testing time, we still use its distribution information as one type of features, since it can facilitate the machine-learning process, and the three groups of features directly map to the three types of prioritization techniques. For each group of features for an instance (i.e., a project with a test suite), PTP ranks them based on the ascending order of their values. Figure 3 shows an example of extracting features for an instance using a project with 3 tests (t_1, t_2, t_3). We first collect test coverage (C), testing time (T), and coverage per unit time (C/T) for each test shown in Figure 3(a), and then rank the values of C, T, and C/T of all tests in ascending order respectively shown in Figure 3(b). Finally, the features of this instance are shown in Figure 3(c). Since different projects tend to have different number of tests that cause the unaligned issue of features, PTP applies zero padding for test coverage, testing time, and coverage per unit time respectively, to obtain the same number of features. Another benefit of zero padding is that such features also consider the impacts of the number of tests on test prioritization, since the number of tests (i.e., test-suite size) may also be an important factor for test distributions.

3.1.2 Instance Labeling. The label for each training instance of PTP is which prioritization technique performs optimal for a project. We use the $APFD_c$ as the measurement of prioritization techniques since it is a more practical measurement than APFD, and compare three types of techniques including the cost-unaware, cost-aware, and cost-only techniques. Here we use the additional technique as the representative of cost-unaware techniques due to the following

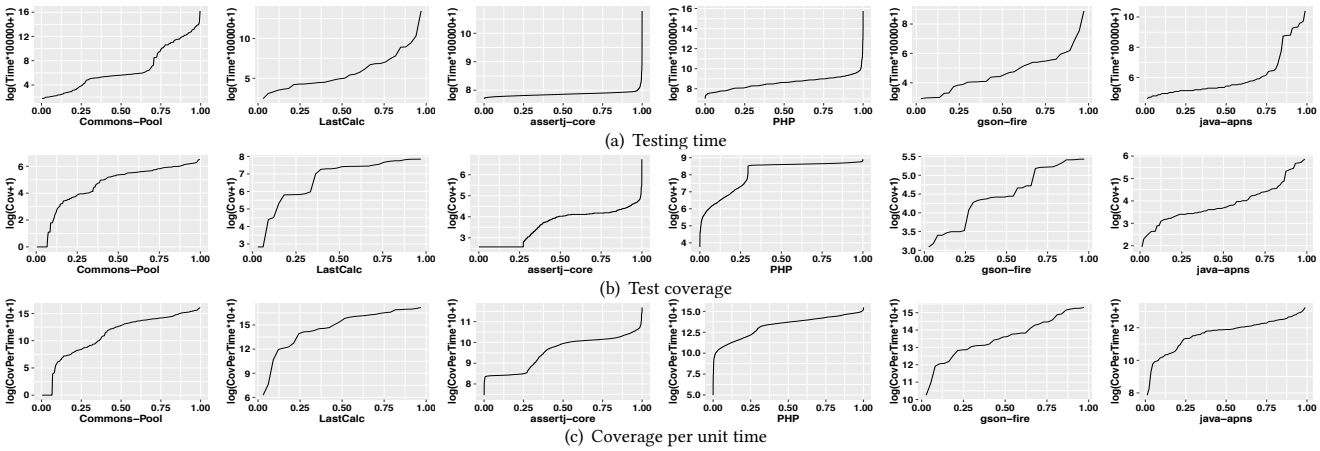


Figure 1: Distribution of testing time, test coverage, and coverage per unit time

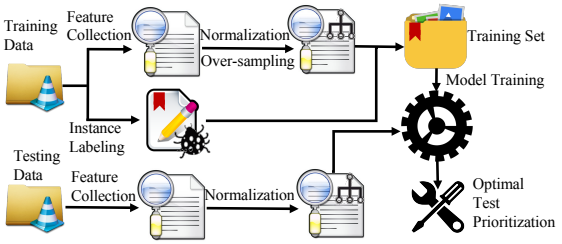


Figure 2: Overview of PTP

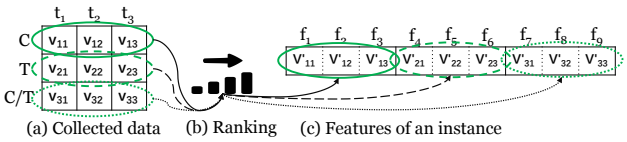


Figure 3: An example of feature collection

two reasons – (1) the cost-aware technique studied in this paper is adapted from the traditional additional technique by Epitropakis et al. [35], and (2) when cost-unaware test prioritization outperforms cost-aware test prioritization and cost-only test prioritization, at least the additional technique will outperform them based on the $APFD_c$ results in Section 2.2.1. Therefore, when collecting the label for an instance, PTP compares the average $APFD_c$ values on all mutation groups for the three techniques, and treats the technique with the highest average $APFD_c$ as the label.

3.1.3 Predictive Model Training. Since different training instances tend to have different value ranges for test coverage, testing time, and coverage per unit time, PTP normalizes the three types of feature values for each training instance into the range [0,1] using min-max normalization [49], respectively. That is, PTP adjusts values measured on different scales into a common scale. For example, supposed the set of values for the coverage feature extracted from a training instance is denoted as $\mathbf{x} = \{\mathbf{x}^{(1)}, \mathbf{x}^{(2)}, \ldots, \mathbf{x}^{(n)}\}$, and the the normalized instance is denoted as $\overline{\mathbf{x}}$ (i.e., value of $\mathbf{x}^{(i)}$ after normalization is denoted as $\overline{\mathbf{x}}^{(i)}$), where $1 \leq i \leq n$. Formula 1 shows the min-max normalization on $\mathbf{x}^{(i)}$. Besides, PTP uses the over-sampling strategy (i.e., resampling the minority class) to deal with the imbalanced data problem following the existing work [17, 43].

$$\overline{\mathbf{x}}_{(i)} = \frac{\mathbf{x}^{(i)} - min(\{\mathbf{x}^{(k)} | 1 \leq k \leq n\})}{max(\{\mathbf{x}^{(k)} | 1 \leq k \leq n\}) - min(\{\mathbf{x}^{(k)} | 1 \leq k \leq n\})} \quad (1)$$

Based on the set of processed training data, PTP trains a predictive model via machine learning, which is used to predict which technique performs optimal for a specific project in practice. The test coverage, testing time, and coverage per unit time of each test are regarded as features, but such basic features are hard to predict the optimal prioritization technique. Based on the observations in the study, the high-level distribution patterns constructed from these basic features are more directly related to the optimal prioritization. The recently proposed XGBoost algorithm [28] is an optimized distributed gradient boosting learning algorithm that can effectively learn high-level features constructed from basic features using tree ensemble models, which is suitable to our problem. Therefore, in PTP we adopt it to learn the high-level patterns. Also, XGBoost supports to automatically do parallel computation to improve efficiency. Note that we also investigate the impact of different machine learning algorithms in Section 3.3.

3.2 PTP Evaluation Design

We conducted an extensive study to evaluate PTP. In particular, this study shares the similar experimental setup as Section 2.1, and we present the differences between them as follows.

3.2.1 Subjects. To make results more generalizable, in this study we scale up our subjects to 50 open-source projects from GitHub, shown in Table 1. Despite that, we just have 50 original test suites for the 50 subjects, the number of prioritization runs is also 50, which is small in number for machine learning. To possess enough instances for PTP, we constructed 50 test suites for each subject by randomly selecting a subset of tests from its original test suite. Each constructed test suite is used as an instance, where test prioritization applies. In this way, we constructed 2,500 such instances in total. Besides, each original test suite can also be used as an instance, and thus we have 2,550 instances in total. As each subject has about 100 mutant groups, for each instance we calculated the average $APFD_c$ values of all the mutant groups as the measurement of the corresponding technique, and labeled the prioritization technique with the largest average $APFD_c$ values. Following the existing work on machine learning [16, 20], we used the *leave-one-out cross-validation* to evaluate PTP. That is, for each subject, we used *all the instances from* ***remaining subjects*** as the training data to predict

the optimal prioritization technique for the ***target subject with its original test suite***.

3.2.2 Measurements. From our industry partners, they tend to care different metrics based on different requirements. For example, they care the time spent on detecting the first regression fault for starting debugging and releasing resources earlier. To sufficiently evaluate the practical effectiveness of PTP, besides $APFD_c$, we used other three practical time-based measurements (in milliseconds), whose importance is confirmed by our industry partners:

- **FT**, the time spent on detecting the first regression fault [48].
- **LT**, the time spent on detecting the last regression fault.
- **AT**, the average time spent on detecting all regression faults. Even though both AT and $APFD_c$ measure the effectiveness of test prioritization by considering all detected faults, they have different computations and can have different results.

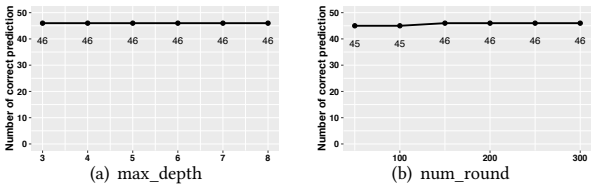
3.2.3 Implementation. We used the XGBoost approach implemented by the XGBoost python package [12] to build the predictive model, choosing the *softmax* objective due to the multiclass classification problem. Based on a preliminary study that we conducted on a small dataset, we set *eta* = 0.1, *max_depth* = 5, *silent* = 1, and *num_round* = 150, and other parameters to be default values. We investigate the impact of parameters in Section 3.3. In this study, we used statement coverage in PTP since it is usually more effective than others [58], which is also confirmed by our study in Section 2.

3.3 Results and Analysis

Overall Effectiveness. Based on experimental results, PTP correctly predicts the optimal technique for 46 of 50 subjects (except *assertj-core*, *asterisk-jave*, *geohash-java*, and *spring-data-solr*), indicating *extremely high accuracy, i.e., 92%*.

Table 6 shows the comparison results of PTP with the other three techniques. Row "Avg" presents the average effectiveness of all subjects, and Row "Imp" presents the average improvement of PTP compared with the other techniques. From the last two rows, PTP outperforms all the three techniques in terms of all time-based measurements. Surprisingly, the improved rate of PTP for FT ranges from 43.16% to 94.92% compared with all existing state-of-the-art techniques. In practice, FT is usually the most important measurement in industry because developers tend to start debugging immediately after the first test failure. That further confirms the practical value of PTP. Moreover, the improved rates of PTP for AT and LT are also high, ranging from 45.30% to 81.23% and 32.84% to 62.59% respectively, demonstrating the practical effectiveness of PTP from various angles. The improved rate of PTP for $APFD_c$ is smaller than other measurements. The reason is that $APFD_c$ has the value range of [0,1], making $APFD_c$ values of different techniques tend to be close.

Parameter Evaluation. We investigate the impact of main parameters in PTP, i.e., *max_depth* and *num_round* for XGBoost [28]. The former represents the maximum depth of a tree, and the latter represents the number of rounds for boosting. Figure 4 shows the number of subjects whose optimal prioritization technique is correctly predicted by PTP when changing each parameter alone. From this figure, regardless of parameter values, the number of subjects predicted correctly is always close to the total number of subjects (i.e., 50), demonstrating the stable effectiveness of PTP.



(a) max_depth (b) num_round

Figure 4: Impact of parameters on PTP

Learning Algorithm Impacts. We investigate the impacts of machine learning algorithms. Besides XGBoost, we use one widely-used baseline algorithm, i.e., Random Guess (RG), and another ten typical classification algorithms, including Multivariate Bernoulli Naive Bayes (BNB) [63], Multi-Layer Perceptron (MLP) [45], Support Vector Machine (SVM) [36], Logistic Regression (LR) [36], Ridge Regression (RR) [46], K-Nearest Neighbors (KNN) [13], Random Forest (RF) [18], Extra-Trees (ET) [38], Linear Discriminant Analysis (LDA) [70], and Quadratic Discriminant Analysis (QDA) [42]. In particular, we used their implementations provided by *scikit-learn* package in Python [11] with their default settings.

Figure 5 shows the comparison results using different machine learning algorithms in terms of effectiveness, i.e., the number of subjects predicted correctly. From this figure, all the machine learning algorithms perform significantly better than the baseline, RG, indicating the promising direction of using machine learning to predict the optimal test prioritization technique. Besides, XGBoost performs better than all other machine learning algorithms, demonstrating the power of XGBoost to predict the optimal test prioritization technique in PTP. The three tree-based algorithms XGBoost, RF, and ET rank at top 3, because these tree-based algorithms are ensemble learning algorithms that integrate multiple decision trees. That indicates our features can be modeled well by tree structures.

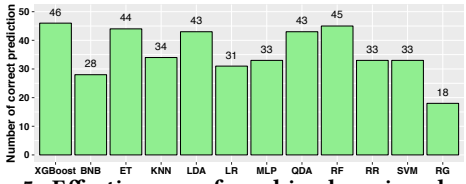


Figure 5: Effectiveness of machine learning algorithms

Table 7 shows the comparison results using different learning algorithms in terms of efficiency (milliseconds), including the average time spent on **offline** training a predictive model and the average time spent on **online** prediction for each subject. We do not show the cost of Random Guess because this algorithm has almost no cost. From this table, the offline training time of XGBoost is larger than many other learning algorithms, but it is still acceptable, i.e., about 4.67 minutes. Moreover, the training process is conducted offline, and thus its cost can be ignored. On the other hand, for all the machine learning algorithms, their online predicting time is negligible, i.e., in milliseconds, demonstrating the practical feasibility of PTP.

Table 7: Efficiency of machine learning algorithms (ms)

Stage	XGBoost	BNB	ET	KNN	LDA	LR	MLP	QDA	RF	RR	SVM
train	279220	936	950	4624	646777	50056	109927	49154	487	4860	818916
predict	10	7	1	51	0	11	12	187	1	1	10

Overall, learning algorithms indeed have impacts on PTP, mostly demonstrated by its effectiveness. The tree-based learning algorithms, especially XGBoost, perform better in terms of effectiveness. Regarding to practical usage, XGBoost is a better choice for PTP.

Table 6: Effectiveness of PTP compared with the other three techniques on open-source subjects

ID	$APFD_c$				FT (ms)				LT (ms)				AT (ms)			
	PTP	Unaware	Aware	Only	PTP	Unaware	Aware	Only	PTP	Unaware	Aware	Only	PTP	Unaware	Aware	Only
1	0.837	0.842	0.837	0.472	793	1012	793	11447	28390	28136	28390	57304	10575	10332	10575	34303
2	0.971	0.738	0.971	0.981	2	644	2	4	567	1906	567	347	137	1201	137	92
3	0.971	0.922	0.971	0.961	77	5864	77	872	197672	357196	197672	221707	42782	113788	42782	56318
4	0.909	0.594	0.909	0.908	886	190869	886	1485	309992	603003	309992	296097	81470	362654	81470	82048
5	0.966	0.532	0.904	0.966	12	58025	7	12	43757	213307	114970	43757	10495	141069	29067	10495
6	0.928	0.550	0.928	0.903	1	24	1	3	34	107	34	37	12	71	12	15
7	0.966	0.826	0.966	0.960	84	10609	84	185	64481	190349	64481	68844	15212	77767	15212	17962
8	0.818	0.799	0.818	0.606	1510	1911	1510	8165	25298	26389	25298	38429	10108	11163	10108	21611
9	0.966	0.576	0.966	0.936	3	414	3	9	244	1276	244	414	70	777	70	120
10	0.995	0.880	0.995	0.995	2	77	2	2	145	3662	145	160	42	960	42	46
11	0.981	0.428	0.805	0.981	7	487332	304	7	71189	644361	507945	71189	17508	540653	162062	17508
12	0.977	0.920	0.973	0.977	50	8150	15	50	26792	78375	36960	26792	7143	25795	8657	7143
13	0.881	0.881	0.774	0.381	49417	49417	90407	186710	251953	251953	321729	752846	110497	110497	207226	561347
14	0.953	0.860	0.953	0.897	163	237	163	2452	12227	36293	12227	15277	3780	11241	3780	8230
15	0.989	0.845	0.977	0.989	17	1230	3	17	3645	29850	7903	3645	836	11710	1746	836
16	0.875	0.792	0.863	0.875	6	646	14	6	7944	9875	8347	7944	2085	3524	2293	2085
17	0.995	0.952	0.993	0.995	30	48	5	30	900	11710	2442	900	335	3500	534	335
18	0.984	0.858	0.968	0.984	10	529	2	10	1396	8497	3163	1396	359	3087	705	359
19	0.976	0.737	0.956	0.976	51	8289	14	51	8356	45228	17010	8356	2125	23130	3877	2125
20	0.988	0.814	0.977	0.988	28	3828	10	28	3881	28520	8494	3881	983	14674	1876	983
21	0.865	0.725	0.865	0.824	24	418	24	11	683	934	683	930	259	582	259	314
22	0.914	0.809	0.914	0.879	75	823	75	95	1296	2001	1296	1515	551	1154	551	708
23	0.779	0.663	0.779	0.611	35	575	35	108	5082	5489	5082	6989	1984	2862	1984	3234
24	0.974	0.607	0.910	0.974	9	10859	4	9	5746	32070	17301	5746	1310	18983	4387	1310
25	0.646	0.603	0.646	0.536	47	65	47	66	362	373	362	384	183	204	183	235
26	0.999	0.945	0.960	0.999	1	1	0	1	26	1623	1163	26	9	356	259	9
27	0.928	0.870	0.928	0.861	0	4	0	1	22	35	22	31	6	12	6	12
28	0.986	0.927	0.982	0.986	2	4	0	2	144	610	200	144	33	153	42	33
29	0.961	0.830	0.971	0.961	1	23	1	1	318	895	260	318	78	308	63	78
30	0.971	0.728	0.951	0.971	69	26116	46	69	23027	111051	45937	23027	6658	59799	11196	6658
31	0.966	0.868	0.966	0.950	6	28	6	7	555	1233	555	709	142	520	142	185
32	0.975	0.746	0.966	0.975	4	283	1	4	428	2226	623	428	110	1102	150	110
33	0.937	0.887	0.937	0.933	3	22	3	5	161	232	161	135	47	85	47	49
34	0.920	0.909	0.920	0.845	8	14	8	28	1484	1578	1484	2207	405	458	405	782
35	0.958	0.786	0.931	0.958	21	375	9	21	1931	6193	3208	1931	527	2604	842	527
36	0.996	0.039	0.870	0.996	20	105149	2	20	2249	107321	49822	2249	576	105997	14500	576
37	0.873	0.861	0.873	0.760	92	432	92	248	7357	7021	7357	12916	2450	2694	2450	4611
38	0.968	0.830	0.968	0.962	2	40	2	3	176	538	176	181	46	247	46	56
39	0.948	0.492	0.948	0.923	312	54924	312	446	55413	196480	55413	67926	14142	135151	14142	20670
40	0.862	0.656	0.862	0.836	14	51	14	56	688	1090	688	533	213	528	213	253
41	0.884	0.797	0.884	0.844	5	50	5	19	1251	1635	1251	1455	331	594	331	440
42	0.962	0.952	0.962	0.871	63	93	63	136	85	120	85	192	78	99	78	178
43	0.985	0.794	0.985	0.985	26	6642	7	26	8894	49628	9240	8894	2113	27976	2015	2113
44	0.883	0.827	0.883	0.863	7	44	7	30	1158	1247	1158	1084	344	511	344	401
45	0.893	0.747	0.893	0.820	4	49	4	17	205	315	205	295	66	156	66	109
46	0.743	0.679	0.751	0.743	19	42	5	19	1020	1085	1018	1020	348	435	337	348
47	0.963	0.795	0.935	0.963	4	137	3	4	576	2111	1022	576	166	875	281	166
48	0.984	0.677	0.956	0.984	6	26494	4	6	10388	74505	25975	10388	2221	43352	5957	2221
49	0.959	0.890	0.959	0.945	2	48	2	5	148	281	148	179	47	130	47	61
50	0.978	0.898	0.959	0.978	2	17	1	2	576	1742	1038	576	133	611	247	133
Avg	0.932	0.764	0.916	0.889	1081	21260	1902	4260	23806	63633	38019	35446	7043	37523	12876	17411
Imp	–	21.99%	1.75%	4.84%	–	94.92%	43.16%	74.63%	–	62.59%	37.38%	32.84%	–	81.23%	45.30%	59.55%

In summary, PTP effectively predicts the optimal test prioritization technique, advancing the practical usage of test prioritization to a large extent. With this approach, practitioners can always have close-to-optimal prioritization results, significantly improving regression testing efficacy in practice.

4 INDUSTRY APPLICATION OF PTP

Recently, PTP has been integrated into the practical testing infrastructure for Baidu, a famous search service provider with over 600M monthly active users. Projects in Baidu have considerable regression testing costs due to the following reasons. First, the projects and tests in Baidu are large-scale. Second, the regression testing process is conducted once a change is submitted according to the continuous integration policy of Baidu, and there is high change frequency. For example, one project in Baidu has about 30 commits per day. Moreover, even though this company has an abundance of resources, the resources are still limited relative to such considerable costs. Therefore, more testing optimizations are in demand. Our PTP approach aims to achieve optimal test prioritization to detect faults earlier, which admirably serves their needs of quicker test feedback and less computing resource consumption, and thus has been integrated into Baidu's practical testing infrastructure for faster fault detection.

PTP does achieve great effectiveness in the practical usage in Baidu. Here we report all the results of PTP on the projects of Baidu collected until Feb. 2018, including 11 industrial subjects, totaling over 3 million of lines of source code, nearly 30K tests, and over 100 hours of testing time. For each subject, we have a set of real regression faults on real faulty versions, to evaluate the effectiveness of PTP. In particular, for these industrial subjects, we collected the real regression faults during practical testing from Dec. 2017 to Feb. 2018, i.e., 86 faulty versions with 499 real regression faults in total. Table 8 shows the basic information of these industrial

Table 8: Industrial subjects from Baidu

ID	SLOC	#Test	Time (ms)	# FV	#Faults
I_1	>200K	2,556	30,225,482	2	10
I_2	>200K	2,546	33,098,270	4	38
I_3	>200K	2,550	31,032,187	5	19
I_4	>500K	4,134	54,176,448	19	144
I_5	>500K	4,128	50,641,444	14	46
I_6	>500K	4,123	51,516,471	3	15
I_7	>500K	4,137	51,674,375	5	17
I_8	>500K	4,139	51,581,491	2	11
I_9	>20K	281	5,887,430	21	150
I_{10}	>20K	299	6,686,943	8	37
I_{11}	>20K	202	2,564,670	3	12

* The last two columns present the number of faulty versions and the number of real regression faults. Due to the policy of Baidu, we hide project names and report the rough scale of SLOC.

subjects and faults. In particular, for each industrial subject, we used the predictive model trained based on the data from the 50 open-source projects used in our work to predict the optimal prioritization technique, and then determined an execution order of tests through the predicted optimal test prioritization. This is a practical application scenario of learning-based techniques, i.e., leveraging sufficient open-source data to facilitate industrial usage.

Based on the results on industrial subjects, PTP correctly predicts the optimal prioritization technique for 9 of 11 subjects, demonstrating the accuracy of 81.82%. Table 9 shows the practical effectiveness of PTP on industrial subjects. Similar to findings on open-source subjects, PTP outperforms all the three techniques in terms of all time-based measurements on industrial subjects. The improved rates of PTP for FT range from 16.18% to 83.79% compared with the three state-of-the-art techniques, demonstrating its practical value in industry. Also, the improved rates of PTP for AT and LT range from 3.22% to 36.51% and from 1.75% to 12.53%. Besides, PTP has improvements in terms of $APFD_c$, ranging from 1.11% to 32.10%.

Even though PTP also achieves great improvements on industrial subjects, we find that the improvements are smaller than those on open-source subjects. One possible reason is that we used mutation faults in open-source subjects while used real faults in industrial subjects. That also reflects that test prioritization techniques perform a bit different on industrial subjects with real faults and open-source subjects with mutation faults, indicating the necessity of using both kinds of subjects to evaluate test prioritization techniques.

Such results indicate that PTP not only achieves significant effectiveness on open-source subjects, but also performs great on industrial subjects with real regression faults. In particular, after applying PTP to Baidu, we received positive feedbacks from the testing team of this company according to the practical usage:

> "*... PTP saves beyond 2X testing costs for the* ***anonymous*** *projects with negligible overheads. ... PTP has been successfully integrated into the practical testing of the* ***anonymous*** *projects ...*"

5 THREATS TO VALIDITY

The threats to *external* validity mainly lie in the subjects and faults. Although these subjects may not sufficiently represent other subjects, we have already used the relatively large number of subjects (50 open-source subjects from GitHub) among existing prioritization studies. Regarding to faults used in the studies, we used mutation faults for open-source subjects, because they are evaluated to be suitable for software testing experimentation [14, 51] and are widely used in test prioritization research [32, 57, 80]. Prior work [65] discussed the threat from mutations, in the future we will reduce this threat accordingly. In this work, to reduce these threats, we also report the results of PTP on industrial subjects with real regression faults, i.e., 11 industrial subjects with 499 real regression faults. Here following prior work [56, 77], we regard each failure as each fault in industrial subjects, since it is hard to distinguish whether different failures are caused by the same fault [77] and many faulty versions in the study have only one failure.

The threats to *construct* validity mainly lie in the regression scenario, instance collection, and measurement. In the studies on open-source subjects, we regard the version without faults as the former version and the version with faults as the current version following test-prioritization literature [59, 80]. To reduce this threat, our industry application of PTP uses the real regression scenario. We do not consider test evolutions here, and in the future we will consider it to further reduce this threat. In particular, the PTP trained model is based on the traditional general test prioritization, which are designed to work for a series of subsequent versions. Two independent recent studies [44, 58] both demonstrated that software changes do not impact the effectiveness of general test prioritization much. In the evaluation of PTP, to possess enough instances, we randomly constructed test suites, besides using the original test suites. However, even based on the set of randomly constructed test suites and original test suites, our study demonstrates that PTP can predict the optimal technique for each project with the original test suite. Considering the difference between constructed test suites and original test suites, we will repeat the experiment by using more real instances. In the evaluation of PTP, we used time-based measurements for test prioritization because PTP can also be viewed as a type of prioritization technique. More discussion on their difference is given by Section 6. However, strictly speaking, PTP is a prediction technique, and thus we will further measure its effectiveness by using more measurements in machine learning.

6 DISCUSSION

•**Practical Implications.** Time plays a non-trivial role in practical test prioritization, demonstrated from at least two aspects: prioritization algorithms and measurements. Regarding to prioritization algorithms, our study shows that the cost-only technique performs surprisingly well on some subjects (e.g., *MapDB*). That is, considering the cost of coverage collection, the cost-only technique may be a good candidate prioritization technique in practice. Regarding to measurements, our study shows that comparison results on APFD are much different from $APFD_c$, indicating the importance of practical factors (e.g., testing time) in test prioritization.

•**New Perspective from PTP.** Our work provides a new perspective for test prioritization problem. In the past, the existing work solves this problem mainly through proposing new prioritization techniques, hoping they always outperform the others. However, due to its inherent difficulties [55], it is quite hard to figure out an outstanding technique, which always produces the best prioritization results. Instead, our work admits the advantages of existing prioritization techniques, and aims at generating the optimal prioritization results for each project through the selection of prioritization techniques. That is, PTP opens an entire new dimension, different from the current research direction of test prioritization.

Table 9: Effectiveness of PTP compared with the other three techniques on industrial subjects

ID	$APFD_c$				FT (ms)				LT (ms)				AT (ms)			
	PTP	Unaware	Aware	Only	PTP	Unaware	Aware	Only	PTP	Unaware	Aware	Only	PTP	Unaware	Aware	Only
I_1	0.808	0.808	0.776	0.216	250787	250787	1293191	14322465	7560276	7560276	7369532	16382085	3798574	3798574	4423204	15457095
I_2	0.705	0.676	0.705	0.726	565043	1474614	565043	4462143	11918502	12917148	11918502	9652705	6491689	7134236	6491689	6026221
I_3	0.821	0.821	0.812	0.317	1240817	1240817	1368700	12457645	7950702	7950702	8496865	16427286	3741602	3741602	3927869	14284088
I_4	0.680	0.658	0.680	0.561	3423653	4784899	3423653	12333858	24271737	24567782	24271737	25505772	13213454	14114248	13213454	18117446
I_5	0.808	0.808	0.762	0.510	4466056	4466056	6505057	18246359	11066564	11066564	13113232	21322579	7607309	7607309	9405948	19361214
I_6	0.658	0.648	0.658	0.442	2661617	3304244	2661617	18050271	21073328	21125779	21073328	29588445	13709244	14126753	13709244	22382345
I_7	0.798	0.772	0.798	0.529	1749060	4398506	1749060	16541650	14262112	14233773	14262112	21330727	8158275	9194197	8158275	19014559
I_8	0.564	0.368	0.564	0.912	790998	15387173	790998	2457853	29316445	33876494	29316445	5842782	17421563	25274566	17421563	3512491
I_9	0.626	0.616	0.626	0.610	546745	597195	546745	747664	3274387	3302778	3274387	3264328	1826911	1879284	1826911	1908097
I_{10}	0.737	0.724	0.737	0.626	649200	649111	649200	2047396	2644144	2734368	2644144	2777201	1663724	1743161	1663724	2364267
I_{11}	0.642	0.622	0.642	0.492	281394	283261	281394	922338	1255995	1329314	1255995	1778476	920840	972075	920840	1301995
Avg	0.713	0.684	0.706	0.540	1511397	3348788	1803151	9326331	12235836	12787725	12454207	13988399	7141199	8144182	7378429	11248165
Imp	–	4.35%	1.11%	32.10%	–	54.87%	16.18%	83.79%	–	4.32%	1.75%	12.53%	–	12.32%	3.22%	36.51%

•**Practical Usage of PTP.** In the evaluation of PTP on open-source subjects, we used the widely-used leave-one-out cross-validation (described in Section 3.2.1), and our results demonstrate the great effectiveness of PTP. Differently, in the industry application of PTP, we used the model trained based on the data of the 50 open-source projects to predict the optimal prioritization technique for industrial subjects directly, and the results further demonstrate the practical effectiveness of PTP. That indicates that PTP can provide generally usable models for its later practical usage, demonstrating the stability and applicability of PTP.

•**Future Extensions of PTP.** For predictive test prioritization, there are a lot of future extensions. First, based on Figure 1, the problem can be transformed to a pattern recognition problem (i.e., image recognition), since the similar distribution images tend to have the same prediction conclusions. In the future, we will collect the distribution images from Github and improve PTP by applying learning or even deep learning techniques to these images. Second, besides the used distributions, some other factors may also influence prioritization technique selection, e.g., the degree of coverage overlapping. In the future, we will leverage such information to improve PTP. Third, PTP uses $APFD_c$ values to label instances, but other time-based measurements like FT, LT, and AT can also be used as labels to guide the selection, depending on developers' specific requirements. In the future, we will extend PTP using various measurements to label instances and evaluate their effectiveness.

7 RELATED WORK

As our work investigates the selection of test prioritization techniques through an empirical study and a novel machine-learning based approach, we summarize the existing work into two parts, test prioritization techniques and empirical studies.

Prioritization Techniques. Most of the existing prioritization techniques belong to the cost-unaware techniques, e.g., the four cost-unaware techniques studied in this paper. Besides, time-aware test prioritization [31, 62, 73, 74, 81] also belongs to this category. Slightly different from the preceding cost-unaware techniques, time-aware techniques focus on prioritizing tests satisfying the time constraint. However, these techniques do not deal with the balance between testing costs and other factors in test prioritization, and thus do not belong to cost-aware techniques. For cost-aware techniques [47, 61, 72, 82], Epitropakis et al. [35] proposed a cost-aware multi-objective prioritization technique, and compared its performance with the cost-cognizant additional greedy technique [61]. Chen et al. [21, 22] proposed a learning-based approach to prioritizing tests for compilers based on the predicted bug-revealing probability per unit time for each test. Busjaeger et al. [19] proposed a learning-based prioritization technique, which prioritizes tests by assigning each test an aggregated score through machine learning. Different from them, our work does not present a prioritization algorithm, but aims at *selecting* the best-performance technique among existing ones for a specific project.

Empirical Studies. Most of existing empirical studies compared the performance among cost-unaware techniques and they usually used the APFD measurement [30, 34, 44, 59, 67, 68]. For example, Hao et al. [41] conducted an empirical study to compare existing coverage-based prioritization techniques in terms of APFD and found that the additional technique even outperforms the optimal coverage-based prioritization technique, which performs the best in terms of coverage (i.e., APXC) rather than fault detection (i.e., APFD). Besides, several empirical studies [35, 61] investigated the performance of cost-aware techniques. In summary, the existing empirical studies investigated only either the performance of cost-unaware techniques or the performance of cost-aware techniques. That is, to our best knowledge, none of the existing work compares the performance of cost-unaware and cost-aware techniques together in terms of practical effectiveness, and this paper is the first one to compare both cost-unaware and cost-aware techniques in terms of actual regression testing time.

8 CONCLUSION

Despite the large body of research on test prioritization, the optimal test prioritization technique in practice still remains unknown. To answer it, this paper performs the first study to compare various prioritization techniques in terms of both APFD and $APFD_c$. The study results reveal a number of practical guidelines, and show that actually no existing technique can always perform the best. Our quantitative and qualitative analyses further show that the distribution of test coverage, testing time and coverage per unit time can serve as the guidelines for selecting the optimal prioritization technique for different cases. Based on the findings, we design a predictive test prioritization approach, PTP, which can predict the optimal prioritization technique for a given project based on its test distribution in prior versions. The experimental results show that PTP outperforms the studied techniques significantly, e.g., by 43.16% to 94.92% in detecting the first regression fault. Furthermore, PTP has been integrated into the testing infrastructure of Baidu, demonstrating 16.18% to 83.79% improvement in detecting the first regression fault compared with state-of-the-art techniques (on 499 real regression faults). The data in this work (except the data on industrial projects due to the policy of the company) are available at our project website: https://github.com/JunjieChen/PTP.

REFERENCES

[1] Accessed: 2018. Camel-core. http://camel.apache.org.
[2] Accessed: 2018. Chukwa. http://chukwa.apache.org.
[3] Accessed: 2018. Clover. https://www.atlassian.com/software/clover.
[4] Accessed: 2018. Commons-Pool. https://commons.apache.org/proper/commons-pool.
[5] Accessed: 2018. Gcov. http://ltp.sourceforge.net/coverage/gcov.php.
[6] Accessed: 2018. HBase. https://hbase.apache.org.
[7] Accessed: 2018. MapDB. http://www.mapdb.org.
[8] Accessed: 2018. OpenTripPlanner. http://www.opentripplanner.org.
[9] Accessed: 2018. PHP. http://php.net.
[10] Accessed: 2018. PIT. http://pitest.org.
[11] Accessed: 2018. scikit-learn. http://scikit-learn.org/stable/.
[12] Accessed: 2018. XGBoost python package. http://xgboost.readthedocs.io/en/latest/python/index.html.
[13] N. S. Altman. 1992. An Introduction to Kernel and Nearest-Neighbor Nonparametric Regression. *American Statistician* 46, 3 (1992), 175–185.
[14] J. H. Andrews, L. C. Briand, and Y. Labiche. 2005. Is Mutation an Appropriate Tool for Testing Experiments?. In *ICSE*. 402–411.
[15] James H. Andrews, Lionel C. Briand, Yvan Labiche, and Akbar Siami Namin. 2006. Using mutation analysis for assessing and comparing testing coverage criteria. *TSE* 32, 8 (2006), 608âĂŞ–624.
[16] Tien-Duy B Le, David Lo, Claire Le Goues, and Lars Grunske. 2016. A learning-to-rank based fault localization approach using likely invariants. In *ISSTA*. 177–188.
[17] Gustavo E. A. P. A. Batista, Ronaldo C. Prati, and Maria Carolina Monard. 2004. A study of the behavior of several methods for balancing machine learning training data. *SIGKDD Explorations* 6, 1 (2004), 20–29.
[18] L. Breiman. 2001. Random Forest. *Machine Learning* 45 (2001), 5–32.
[19] Benjamin Busjaeger and Tao Xie. 2016. Learning for test prioritization: an industrial case study. In *FSE*. 975–980.
[20] Gavin C Cawley and Nicola LC Talbot. 2003. Efficient leave-one-out cross-validation of kernel fisher discriminant classifiers. *Pattern Recognition* 36, 11 (2003), 2585–2592.
[21] Junjie Chen. 2018. Learning to accelerate compiler testing. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*. 472–475.
[22] Junjie Chen, Yanwei Bai, Dan Hao, Yingfei Xiong, Hongyu Zhang, and Bing Xie. 2017. Learning to Prioritize Test Programs for Compiler Testing. In *ICSE*. 700–711.
[23] Junjie Chen, Yanwei Bai, Dan Hao, Yingfei Xiong, Hongyu Zhang, Lu Zhang, and Bing Xie. 2016. Test case prioritization for compilers: A text-vector based approach. In *ICST*. 266–277.
[24] Junjie Chen, Yanwei Bai, Dan Hao, Lingming Zhang, Lu Zhang, and Bing Xie. 2017. How do assertions impact coverage-based test-suite reduction?. In *ICST*. 418–423.
[25] Junjie Chen, Yanwei Bai, Dan Hao, Lingming Zhang, Lu Zhang, Bing Xie, and Hong Mei. 2016. Supporting oracle construction via static analysis. In *ASE*. 178–189.
[26] Junjie Chen, Wenxiang Hu, Lingming Zhang, Dan Hao, Sarfraz Khurshid, and Lu Zhang. 2018. Learning to accelerate symbolic execution via code transformation. In *ECOOP*, Vol. 109. 6:1–6:27.
[27] Lingchao Chen and Lingming Zhang. 2018. Speeding up Mutation Testing via Regression Test Selection: An Extensive Study. In *ICST*. 58–69.
[28] Tianqi Chen and Carlos Guestrin. 2016. Xgboost: A scalable tree boosting system. In *Proceedings of the 22Nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. 785–794.
[29] Henry Coles, Thomas Laurent, Christopher Henard, Mike Papadakis, and Anthony Ventresque. 2016. PIT: a practical mutation testing tool for Java. In *ISSTA*. 449–452.
[30] H. Do, S. Mirarab, L. Tahvildari, and G. Rothermel. 2008. An empirical study of the effect of time constraints on the cost-benefits of regression testing. In *FSE*. 71–82.
[31] Hyunsook Do, Siavash Mirarab, Ladan Tahvildari, and Gregg Rothermel. 2010. The effects of time constraints on test case prioritization: A series of controlled experiments. *IEEE Transactions on Software Engineering* 36, 5 (2010), 593–617.
[32] Hyunsook Do and Gregg Rothermel. 2006. On the use of mutation faults in empirical assessments of test case prioritization techniques. *TSE* 32, 9 (2006), 733–752.
[33] Sebastian Elbaum, Alexey Malishevsky, and Gregg Rothermel. 2001. Incorporating varying test costs and fault severities into test case prioritization. In *ICSE*. 329–338.
[34] S. Elbaum, A. Malishevsky, and G. Rothermel. 2002. Test case prioritization: A family of empirical studies. *TSE* 28, 2 (2002), 159 – 182.
[35] Michael G Epitropakis, Shin Yoo, Mark Harman, and Edmund K Burke. 2015. Empirical evaluation of pareto efficient multi-objective regression test case prioritisation. In *ISSTA*. 234–245.
[36] Rong-En Fan, Kai-Wei Chang, Cho-Jui Hsieh, Xiang-Rui Wang, and Chih-Jen Lin. 2008. LIBLINEAR: A Library for Large Linear Classification. *JMLR* 9 (2008), 1871–1874.
[37] Qing Gao, Jun Li, Yingfei Xiong, Dan Hao, Xusheng Xiao, Kunal Taneja, Lu Zhang, and Tao Xie. 2016. High-confidence software evolution. *Science China Information Sciences* 59, 7 (2016), 071101.
[38] Pierre Geurts, Damien Ernst, and Louis Wehenkel. 2006. Extremely randomized trees. *Machine Learning* 63, 1 (2006), 3–42.
[39] Milos Gligoric, Lamyaa Eloussi, and Darko Marinov. 2015. Practical regression test selection with dynamic file dependencies. In *ISSTA*. 211–222.
[40] Dan Hao, Lu Zhang, and Hong Mei. 2016. Test-case prioritization: achievements and challenges. *FCS* 10, 5 (2016), 769–777.
[41] Dan Hao, Lu Zhang, Lei Zang, Yanbo Wang, Xingxia Wu, and Tao Xie. 2016. To Be Optimal Or Not in Test-Case Prioritization. *TSE* 42, 5 (2016), 490–505.
[42] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. 2001. The elements of statistical learning. 2001. *Springer* 167, 1 (2001), 192–192.
[43] Haibo He and Edwardo A. Garcia. 2009. Learning from Imbalanced Data. *TKDE* 21, 9 (2009), 1263–1284.
[44] Christopher Henard, Mike Papadakis, Mark Harman, Yue Jia, and Yves Le Traon. 2016. Comparing white-box and black-box test prioritization. In *ICSE*. 523–534.
[45] Geoffrey E Hinton. 1989. Connectionist learning procedures. *Artificial intelligence* 40, 1 (1989), 185–234.
[46] Arthur E. Hoerl and Robert W. Kennard. 1970. Ridge Regression: Biased Estimation for Nonorthogonal Problems. *Technometrics* 12, 1 (1970), 55–67.
[47] Yu-Chi Huang, Kuan-Li Peng, and Chin-Yu Huang. 2012. A history-based cost-cognizant test case prioritization technique in regression testing. *Journal of Systems and Software* 85, 3 (2012), 626–637.
[48] Vilas Jagannath, Yun Young Lee, Brett Daniel, and Darko Marinov. 2009. Reducing the costs of bounded-exhaustive testing. In *FASE*. 171–185.
[49] Y Kumar Jain and Santosh Kumar Bhandare. 2011. Min max normalization based data perturbation method for privacy protection. *International Journal of Computer & Communication Technology* 2, 8 (2011), 45–50.
[50] Bo Jiang, Zhenyu Zhang, Wing Kwong Chan, and TH Tse. 2009. Adaptive random test case prioritization. In *ASE*. 233–244.
[51] René Just, Darioush Jalali, Laura Inozemtseva, Michael D. Ernst, Reid Holmes, and Gordon Fraser. 2014. Are Mutants a Valid Substitute for Real Faults in Software Testing?. In *FSE*. 654–665.
[52] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. 2012. Genprog: A generic method for automatic software repair. *TSE* 38, 1 (2012), 54–72.
[53] Owolabi Legunsen, Farah Hariri, August Shi, Yafeng Lu, Lingming Zhang, and Darko Marinov. 2016. An extensive study of static regression test selection in modern software evolution. In *FSE*. 583–594.
[54] Z. Li and M. Harman andR. Hierons. 2007. Search algorithms for regression test case prioritisation. *TSE* 33, 4 (2007), 225–237.
[55] Zheng Li, Mark Harman, and Robert M Hierons. 2007. Search algorithms for regression test case prioritization. *TSE* 33, 4 (2007), 225–237.
[56] Christopher Lidbury, Andrei Lascu, Nathan Chong, and Alastair F Donaldson. 2015. Many-core compiler fuzzing. *PLDI* 50, 6 (2015), 65–76.
[57] Yiling Lou, Dan Hao, and Lu Zhang. 2015. Mutation-based test-case prioritization in software evolution. In *ISSRE*. 46–57.
[58] Yafeng Lu, Yiling Lou, Shiyang Cheng, Lingming Zhang, Dan Hao, Yangfan Zhou, and Lu Zhang. 2016. How Does Regression Test Prioritization Perform in Real-world Software Evolution?. In *ICSE*. 535–546.
[59] Qi Luo, Kevin Moran, and Denys Poshyvanyk. 2016. A Large-scale Empirical Comparison of Static and Dynamic Test Case Prioritization Techniques. In *FSE*. 559–570.
[60] Qi Luo, Kevin Moran, Lingming Zhang, and Denys Poshyvanyk. [n. d.]. How Do Static and Dynamic Test Case Prioritization Techniques Perform on Modern Software Systems? An Extensive Study on GitHub Projects. ([n. d.]).
[61] Alexey G Malishevsky, Joseph R Ruthruff, Gregg Rothermel, and Sebastian Elbaum. 2006. Cost-cognizant test case prioritization. *Department of Computer Science and Engineering, University of Nebraska-Lincoln, Techical Report* (2006).
[62] Dusica Marijan, Arnaud Gotlieb, and Sagar Sen. 2013. Test case prioritization for continuous regression testing: An industrial case study. In *Software Maintenance (ICSM), 2013 29th IEEE International Conference on*. IEEE, 540–543.
[63] Andrew McCallum and Kamal Nigam. 1998. A comparison of event models for Naive Bayes text classification. In *AAAI*. 41–48.
[64] Hong Mei, Dan Hao, Lingming Zhang, Lu Zhang, Ji Zhou, and Gregg Rothermel. 2012. A static approach to prioritizing junit test cases. *TSE* 38, 6 (2012), 1258–1275.
[65] Mike Papadakis, Christopher Henard, Mark Harman, Yue Jia, and Yves Le Traon. 2016. Threats to the validity of mutation-based test assessment. In *ISSTA*. 354–365.
[66] Gregg Rothermel and Mary Jean Harrold. 1996. Analyzing regression test selection techniques. *TSE* 22, 8 (1996), 529–551.
[67] Gregg Rothermel, Roland H Untch, Chengyun Chu, and Mary Jean Harrold. 1999. Test case prioritization: An empirical study. In *ICSM*. 179–188.
[68] Gregg Rothermel, Roland H. Untch, Chengyun Chu, and Mary Jean Harrold. 2001. Prioritizing test cases for regression testing. *TSE* 27, 10 (2001), 929–948.

[69] Ripon K Saha, Lingming Zhang, Sarfraz Khurshid, and Dewayne E Perry. 2015. An information retrieval approach for regression test prioritization based on program changes. In *Proceedings of the 37th International Conference on Software Engineering-Volume 1*. 268–279.
[70] Theofanis Sapatinas. 2010. Discriminant Analysis and Statistical Pattern Recognition. *JRSS* 168, 3 (2010), 635–636.
[71] August Shi, Alex Gyori, Milos Gligoric, Andrey Zaytsev, and Darko Marinov. 2014. Balancing Trade-offs in Test-suite Reduction. In *FSE*. 246–256.
[72] Helge Spieker, Arnaud Gotlieb, Dusica Marijan, and Morten Mossige. 2017. Reinforcement Learning for Automatic Test Case Prioritization and Selection in Continuous Integration. In *ISSTA*. 12–22.
[73] Bharti Suri and Shweta Singhal. 2011. Analyzing test case selection & prioritization using ACO. *ACM SIGSOFT Software Engineering Notes* 36, 6 (2011), 1–5.
[74] Kristen R Walcott, Mary Lou Soffa, Gregory M Kapfhammer, and Robert S Roos. 2006. Timeaware test suite prioritization. In *ISSTA*. 1–12.
[75] Song Wang, Jaechang Nam, and Lin Tan. 2017. QTEP: quality-aware test case prioritization. In *ESEC/FSE*. 523–534.
[76] Thomas H Wonnacott and Ronald J Wonnacott. 1972. *Introductory statistics*. Vol. 19690.
[77] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and understanding bugs in C compilers. In *PLDI*, Vol. 46. 283–294.
[78] Shin Yoo and Mark Harman. 2012. Regression testing minimization, selection and prioritization: a survey. *STVR* 22, 2 (2012), 67–120.
[79] Lingming Zhang. 2018. Hybrid regression test selection. In *ICSE*. 199–209.
[80] Lingming Zhang, Dan Hao, Lu Zhang, Gregg Rothermel, and Hong Mei. 2013. Bridging the gap between the total and additional test-case prioritization strategies. In *ICSE*. 192–201.
[81] Lu Zhang, Shan-Shan Hou, Chao Guo, Tao Xie, and Hong Mei. 2009. Time-aware test-case prioritization using integer linear programming. In *ISSTA*. 213–224.
[82] Xiaofang Zhang, Changhai Nie, Baowen Xu, and Bo Qu. 2007. Test case prioritization based on varying testing requirement priorities and test case costs. In *Quality Software, 2007. QSIC'07. Seventh International Conference on*. IEEE, 15–24.
[83] Yucheng Zhang and Ali Mesbah. 2015. Assertions are strongly correlated with test suite effectiveness. In *FSE*. 214–224.