How Different is Test Case Prioritization for Open and Closed Source Projects?

Xiao Ling, Rishabh Agrawal, and Tim Menzies, *Fellow, IEEE*

Abstract—Improved test case prioritization means that software developers can detect and fix more software faults sooner than usual. But is there one "best" prioritization algorithm? Or do different kinds of projects deserve special kinds of prioritization? To answer these questions, this paper applies nine prioritization schemes to 31 projects that range from (a) highly rated open-source Github projects to (b) computational science software to (c) a closed-source project. We find that prioritization approaches that work best for open-source projects can work worst for the closed-source project (and vice versa). From these experiments, we conclude that (a) it is ill-advised to always apply one prioritization scheme to all projects since (b) prioritization requires tuning to different project types.

Index Terms—software testing, regression testing, test case prioritization, open-source software

1 Introduction

Regression testing is widely applied in both open-source projects and closed-source projects [1]–[3]. When software comes with a large regression suite, then developers can check if their new changes damage old functionality.

Excessive use of regression testing can be expensive and time consuming, especially if run after each modification to software. Such high-frequency regression testing can consume as much as 80 percent of the testing budget, and require half the software maintenance effort [4].

To reduce the cost of performing regression testing, *test case prioritization (TCP)* is widely studied in software testing [1], [2], [5]–[9]. In this approach, some features are extracted from prior test suites and test results and then applied to prioritize the current round of tests. Google reports that test case prioritization can reduce the time for programmers to find 50% of the failing tests from two weeks to one hour [9].

Prioritization schemes that work on some projects may fail on others. As shown later in this paper, not all projects track the information required for all the different prioritization algorithms. For example, suppose closed-source projects are prioritized using the textual descriptions of the test cases. That approach may not always work for open-source projects where such textual descriptions may be absent. Previously, Yu et al. [10] reported that the TERMINATOR test case prioritization algorithm was better than dozens of alternatives. However, TERMINATOR was developed for closed-source proprietary software. This raises the question: does TERMINATOR work for other kinds of projects (e.g. open-source projects)?

To explore this issue, this paper applies prioritization test case schemes to data from a closed-source proprietary project and 30 open-source projects. To the best of our knowledge, this study explores more prioritization algorithms applied to more kinds of data than prior work. Using that data, we answer the following research questions.

RQ1: What is the best algorithm for the closed-source project? We find that we can reproduce prior results:

> *As seen before, the TERMINATOR prioritization scheme works best for that closed-source project.*

RQ2: What is the best algorithm for open-source projects? While our **RQ1** results concurred with past work, **RQ2** shows that closed-source prioritization methods should not be applied to open-source projects:

> *For open-source projects, the best approach is not TERMINATOR, but rather to prioritize using either passing times since last failure or another exponential metric (defined in §3.4).*

RQ3: Do different prioritization algorithms perform various in the open-source projects and the closed-source project? Combining RQ1 and RQ2, we can assert:

> *Test case prioritization schemes that work* best *for the industrial closed-source project can work worse for open-source projects (and vice versa).*

The rest of this paper is structured as follows. Section 2 describes related work and Section 3 explains our experimental methods. Section 4 shows answers to the above questions. This is followed by some discussion in Section 5 and a review of threats to validity in Section 6, Section 7 shows our conclusion which is:

> *It is ill-advised to always apply one prioritization scheme to all projects since prioritization requires tuning to different projects types.*

To say that another way, prioritisation schemes should always be re-assessed using local data. To simplify that process, we have made available on-line all the scripts and data used in this study[1]. Note that those scripts include all the major prioritization schemes seen in the current literature.

• *X. Ling, R. Agrawal and T. Menzies are with the Department of Computer Science, North Carolina State University, Raleigh, USA. E-mail: lingxiaohzsz3ban@gmail.com, ragrawa3@ncsu.edu, timm@ieee.org*

[1] https://github.com/ai-se/TCP2020

2 Background

2.1 Definitions

This paper shows that the "best" prioritization differs between closed-source proprietary projects and open-source projects. These projects can be distinguished as follows:

- **Open-source projects** are developed and distributed for free redistribution, possibility for modifications, and with full access to the source code [11], [12].
- **Closed-source projects** are proprietary software, developed with authorized users with private modification, republishing under a permission agreement [13].

As to the sites where we collect data:

- **Github** is a hosting company for software development version control. Free GitHub accounts are commonly used to host open-source projects. As of January 2020, GitHub reports having over 40 million users and more than 100 million repositories (including at least 28 million public repositories), making it the largest host of source code in the world.
- **TravisTorrent** is a public data set containing vanilla API data (build information), the build log analysis (tests information), plus repository and commit data [14].
- **Travis CI** is an OSS continuous integration as-a-service platform that can run the test suits automatically after each of the commit in the GitHub [15].

2.2 Why Study Test Case Prioritization?

In software development, regression testing is very important in detecting software faults. However, it is also widely recognized as an expensive process. The most helpful approach to reduce computational cost and potentially reveal faults earlier is called test case prioritization [17], [18], [27]–[29]. Better test case prioritization is useful since:

- When developers extend a code base, they can check that their new work does not harm existing functionality.
- This, in turn, enables a rapid release process where developers can safely send new versions of software to users each week (or even each day).
- Faults can be revealed earlier than normal execution, which significantly increases the efficiency and reduces the cost of regression testing. Moreover, within a time limit, more faults can be detected by performing test cases prioritization [1], [2], [24], [30].
- Test managers can locate and fix faults earlier than normal execution by applying test cases prioritization [31].

There are many scenarios where the test case prioritization results of this paper can be applied. According to Zemlin [32], 80 percent of current software projects are open-source projects. Some projects even have extensive test suites. To maintain the stability of projects, project developers want to detect more faults in limited time after each modification. For that purpose, test case prioritization is widely applied in regression testing [1], [2], [5]–[9]. Therefore, a well-performed prioritization algorithm for open-source projects is highly demanded, which can let project developers:

- Detect more faults within a period of time.
- Start to fix software bugs earlier than usual.

Moreover, test case prioritization is applied in the industrial closed-source projects. For example, LexisNexis is an industrial company that provides legal research, risk management, and business research [33]. The Lexis Advance platform is maintained by a set of automated UI tests, which is a case of regression testing. Such testing tasks are very expensive in execution time. Yu et al. state that the automated UI test suite that LexisNexis uses on testing takes approximate 30 hours to execute [10]. Therefore, LexisNexis seeks a prioritization algorithm that can help developers to

1) Test software more often, then ship more updates to customers, at a faster rate;
2) Save time when waiting for feedback on the last change [10].

2.3 Who studies Test Case Prioritization?

For all the above reasons, many researchers explore test case prioritization approach. For example:

- Yu et al. introduced an active learning based framework called TERMINATOR, which implements a Support Vector Machine classifier to achieve higher fault detection rates on automated UI testing [10].
- Hemmati et al. propose a risk-driven clustering method that assigns the highest risk to the tests that failed in the closest version before the current version. After that, tests that failed in the two versions before the current version will be assigned and so on [25].
- Fazlalizadeh et al. propose a test case fault detection performance approach which calculates the ratio of the number of times that the execution of the test case fails to the number of executions of the test case [1].
- Kim et al. claim that the selection probabilities of each test case at each test run is useful in prioritization. They propose an "Exponential Decay Metric" (defined later in this paper) which can calculate selection probabilities with weighted individual history observation [19].
- Zhu et al. and Cho et al. study the correlations between two test cases. They introduce different test case prioritization approaches based on different information on correlation. Zhu et al. purpose co-failure distributions, while Cho et al. implement the flipping history of two test cases [8], [26]. Based on Cho et al., two test cases are "flipped" if they change to the opposite status (Pass or Fail) in two consecutive runs [8].
- Li et al study five search techniques (Hill Climbing, Genetic Algorithm, Greedy, Additional Greedy, and 2-Optimal Greedy) for code coverage [18].
- Elbaum et al use four approaches with function coverage information of test cases. They point out that different testing scenarios should apply the appropriate prioritization approach [34].
- For more examples, see [20], [28], [29], [31], [35]–[37].

2.4 How to Study Test Case Prioritization?

In order to base this work on current methods in the literature, we base this paper on two literature reviews of test case prioritization. In March 2019, Yu et al. explored 1033 papers by using incremental text mining tools and found a list of prioritization algorithms that covered the

TABLE 1
Summary of literature. "#Scheme" shows number of prioritization methods studied. "# Closed" and "# Open" shows how much data was used (measured in terms of number of projects).

		# Scheme	# Closed	# Open	Year	Venue	Citations
Prioritizing Test Cases For Regression Testing	[16]	9	0	8	2001	TSE	1345
Test case prioritization: A family of empirical studie	[17]	18	0	8	2002	TSE	994
Search algorithms for regression test case prioritization	[18]	5	0	6	2007	TSE	739
A history-based test prioritization technique .	[19]	1	0	8	2002	ICSE	461
Adaptive random test case prioritization	[20]	9	0	11	2009	ASE	222
System Test Case Prioritization of New and Regression Test Case	[21]	1	0	0	2005	ESEM	223
Techniques for improving regression testing in continuous integration...	[9]	1	1	0	2014	FSE	187
A clustering approach to improving test case prioritization.	[22]	1	1	0	2011	ICSM	97
Test case prioritization for black box testing	[23]	2	2	0	2007	COMPSAC	94
Test case prioritization for continuous regression testing.	[24]	1	1	0	2013	ICSM	87
Prioritizing test cases for resource constraint environments...	[1]	2	0	7	2009	ICCSIT	32
Prioritizing manual test cases in traditional & rapid release environment	[25]	3	0	1	2015	ICST	30
History-based test case prioritization for failure information	[8]	1	0	2	2016	APSEC	11
Test re-prioritization in continuous testing environment	[26]	1	2	0	2018	ICSME	10
This paper ⟹		9	1	30	2020	TSE	

previous outperformed test case prioritization methods in this area [10]. To confirm and extend that finding for different types of projects, in May 2020, we conducted our own review. Beginning with papers from senior SE venues (as defined by Google Scholar Metrics "software systems"), we searched for highly cited or recent papers studying *test case prioritization*. For our purposes, "highly cited" means at least 10 citations per year since publication. This search found a dozen high profile test prioritization papers in the last 10 years. To that list, we used our domain knowledge to add two paper that we believed to be the most influential early contributions to this work. The final list of 14 papers is Table 1.

Based on the papers in Table 1, and the study of Yu et al, we find that the following history-based information is usually used in test case prioritization. We exclude coverage-based algorithms in this work because (a) collecting proper coverage information for each build in our open-source projects is not only hard, but also time-consuming and (b) our proprietary project from our industrial partner is private. We cannot access to the coverage information. Note that any term *in italics* is defined later in this paper (see §3.4).

- *Time since last failure*: Prioritize test cases by using the numbers of consecutive non-failure [9], [25].
- *Failure rate*: Prioritize test cases by the ratio of total failure times over total execution times [1].
- *Exponential Decay Metrics*: Prioritize test cases by applying Exponential Decay Metrics, which adds weights in execution history [19].
- *ROCKET Metrics*: Prioritize test cases by applying ROCKET 4 Metrics [24].
- *Co-failure*: Prioritize test cases by Co-failure distribution information [26].
- *Flipping History*: Prioritize test cases by the correlations of flipping history [8].
- *TERMINATOR*: An active learning method [10].

For our study, we implement the above algorithms to discover the best approach for the closed-source project and the open-source projects.

TABLE 2
Sanity Check. From [38]

Test	Criteria
Developers	$>= 7$
Pull Requests	> 0
Commits	> 20
Releases	> 1
Issues	> 10
Duration	> 1 year
Has Travis CI	True
Total Builds	$>= 500$
Useful Builds	$>= 100$
Failed Test Cases	$>= 50$

3 Methods

Our overall experimental framework is described in Figure 1. This section offers details on that framework.

3.1 Data Collection

For closed-source project, we use the data set from Yu et al. [10]. For open-source projects, we searched GitHub. Many projects in GitHub are very small or are out of maintenance, which may not have enough information for our experiments. To avoid these

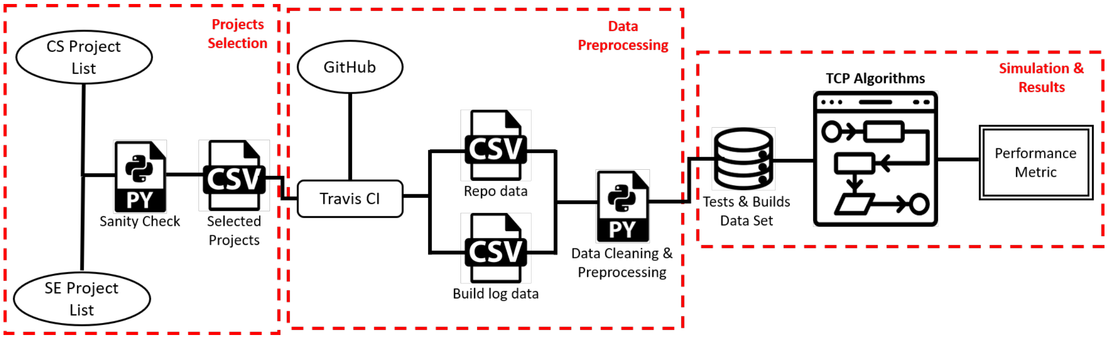

Fig. 1. Framework for our Experiment

TABLE 3
Summary of projects used in this study. (IQR = (75-25)th percentile).

Feature	Min	Median	IQR	Max
Developers	8	39	57	188
Commits	2658	6189	10067	43627
Releases	1	15	21	167
Issues	310	827	667	3047
Duration (week)	137	292	197	529
Total Builds	758	5094	5017	24692
Useful Builds	193	719	991	7579
Failed Test Cases	74	530	261	3554

(a) Summary of 10 CS projects

Feature	Min	Median	IQR	Max
Developers	24	124	258	4020
Commits	969	14446	20939	77152
Releases	23	95	179	426
Issues	192	2369	2882	13848
Duration (week)	342	470	81	636
Total Builds	206	2703	2597	19447
Useful Builds	117	262	406	8794
Failed Test Cases	50	93	111	5517

(b) Summary of 20 SE projects

traps, we implement the GitHub "sanity check", which is introduced in the literature [38]–[40]. Our selection criteria is shown in Table 2. Most of GitHub conditions in Table 2 are straight forward, but the last four conditions need explanation:

- **Has Travis CI:** We use the Travis CI API for collecting repository and build log information. Travis CI can let project developers test their applications and record testing information. Therefore, our ideal projects must implement Travis CI for the testing purpose.
- **Total Builds:** We use Travis CI to collect the execution history of test cases in each build. Travis will automatically trigger test suites after each build. However, for some builds that software developers skip the Travis manually, we discard these builds because they do not trigger the test suits. The number of rest builds are called *total builds*. This sanity check criteria helps us to avoid small projects.
- **Useful Builds:** Among total builds, there are three types of builds. First is the passing builds. Since we aim to prioritize failed test cases, we ignore these builds in this study because they have no failed test cases. Ignoring these builds will not change the orders of test cases. Second is the broken builds with no tests failed. We discard these broken builds since they did not fail due to the execution of test suites. Third is the broken builds with tests failed. These builds are called "failing builds", and are marked as useful builds in this study. This sanity check criteria helps us to avoid projects that do not have enough information for this study.
- **Failed Test Cases:** We count all failed test cases in the entire project. If a project has a very small number of failed test cases, then such a project is not suitable for our experiments.

In order to ensure a diversity of open-source projects, we divided the projects found in this way into different populations:

- We explored the "usual suspects"; i.e. projects that satisfy the sanity checks of Table 2. Note that many of these projects have been used before in other publications. We call this first group the general software engineering group (hereafter, SE).
- We also explored software from the computational science community. Computational Science (hereafter, CS) field studies and develops software to explore astronomy, astrophysics, chemistry, economics, genomics, molecular biology, oceanography, physics, political science, and many engineering fields [38].

After the above analysis, we find ten projects from computational science and twenty projects from software engineering that suitable for our analysis: see Table 3.

As to our closed-source case study data, this is the same data used in Yu et al. case study [10]. For reasons of corporate confidentiality, detailed information is not publicly available. However, we can make some general comments:

- Our closed-source data comes from the nightly regression test suites which executed by LexisNexis. This data is from dozens of projects, with hundreds of developers, working in multiple locations around the globe, all using the same cloud-based testing service.
- LexisNexis is a corporation providing computer-assisted legal research (CALR) as well as business research and risk management services [41], [42]. LexisNexis provides regulatory, legal, business information and analytic to the legal community. As of 2006, LexisNexis company had the world's largest electronic database for legal and public-records related information [43].
- The LexisNexis platform is maintained by a set of automated UI test suites. Those test suites simulate user behaviors on the interface of the platform and detect potential failures of the underlying micro-services whenever the system is modified and rebuilt [10].

3.2 Data Preprocessing

We used the Travis CI API to extract GitHub repository information, (such as unique build id, commit id, and the starting time of the build), and build log information, (such as unique build id, build log status, and failed test cases). By matching the unique id in these two data sets, we can form our experimental data set. Since our target is prioritizing test cases in the new build by using previous execution history, the order of builds by time matters in this experiment. Fortunately, in most cases, Travis CI API will return test builds in the consecutive order. Therefore, we only need to make small modifications to the data we collected. After we obtained information on failed test cases and test builds, we used a Python script to transfer the repository data and the build log data to the build-to-test tests record table for each project.

TABLE 4
Information of Test Case Prioritization Algorithms

Group ID	Information Utilized	Algorithm	Algorithm Description
A	None	A1	Prioritize test cases randomly.
		A2	Prioritize test cases optimally.
B	Execution History	B1	Prioritize test cases by the information of time since last failure.
		B2	Prioritize test cases by the failure rate.
		B3	Prioritize test cases by Exponential Decay Metrics.
		B4	Prioritize test cases by ROCKET Metrics.
C	Execution History, Feedback Information	C1	Prioritize test cases by co-failure information.
		C2	Prioritize test cases by flipping history.
D	Execution History, Feedback Information	D1	Prioritize test cases by TERMINATOR with execution history feature.

3.3 Performance Metric

For the evaluation of prioritization algorithms, we implement fault detection rates. Rothermel et al. [16] state that improved fault detection rate provides feedback faster than usual, which allows developers to correct faults earlier than normal time [16]. Their preferred measurement is called the weighted average of the percentage of faults detected (APFD). APFD calculates the area inside the curve that interpolates the gain in the percentage of detected faults [16]. It is calcuated as follows:

$$APFD = 1 - \frac{TC_1 + TC_2 + \cdots + TC_m}{nm} + \frac{1}{2n} \qquad (1)$$

where:

- TC_i: The rank i of the test case after prioritization that reveals fault.
- m: Total number of faults that are revealed in current test run.
- n: Total number of test cases in the current test run.

APFD ranges from 0 percent to 100 percent. A higher APFD value represents a larger area under the curve, which means higher fault detection rate, or better test case prioritization.

In APFD, all test cases are presumed to have the same execution time. Since the cost of test cases in GitHub projects is hard to be collected, APFD is the most suitable performance metric in our experiment.

3.4 Test Case Prioritization Algorithms

Our study implements the nine prioritization algorithms found in the literature review of §2.4. While all these rely on execution history, they prioritize test cases in different ways. We group these algorithms into Group A, B, C, and D according to the kinds of information that they use.

- **Group A**: Group A contains 2 approaches that prioritize test cases with no information gain. These two algorithms are baseline methods that are used for comparison.
- **Group B**: Group B includes 4 approaches that prioritize test cases only by their own execution history. They sort metrics to reorder the test cases before each test run.
- **Group C**: Group C has 2 approaches that prioritize test cases by correlations between two test cases. Two test cases have a large probability to have the same outcomes if they are highly correlated.
- **Group D**: Group D contains the proposed active learning based framework TERMINATOR [10]. TERMINATOR trains the SVM model with execution history when the first fault is detected.

Table 4 shows the detailed group division and a brief description of each algorithm. The **information utilized** shows what history details is used by each algorithms.

Before we enter the detailed explanation of each algorithm we explored, there are some fundamental test case prioritization terms used in this study should be clarified.

- In this study, a "test run" is defined as a single useful build in the projects. We start to record the APFD score for each test run after the fifth test run as Yu et al did in their work [10]. This indicates that a project with n useful builds will have $n - 5$ recorded APFD scores.
- For each test run, test case prioritization schemes will use all previous useful builds as the indicators to prioritize test cases.

In the rest of this section, we will explain how each algorithm prioritize test cases in each test build. To clearly illustrate how each algorithm works, we construct two small version of test case tables in Table 5, which have four test cases (T_1 - T_4) and five executed test builds (B_1 - B_5) in each example. Our target in these examples are prioritizing test cases for the sixth build. We put two examples here for two reasons

- Some algorithms may result same orders in the small example. Difference can be distributed by comparing two examples.
- Most of the test cases in example A have different number of failures in 5 builds, while most of the test cases in example B have similar number of failures, but failures are in different builds.

In these tables, ✘ indicates failed testing result, and ✔ indicates passed testing result. We assume all test cases have the same cost in our study.

TABLE 5
Examples for Prioritization Algorithms

Test Case	B_1	B_2	B_3	B_4	B_5	B_6
T_1	✘					✔
T_2	✘	✘		✘		✘
T_3		✘	✘			✔
T_4		✘		✘	✘	✘

(a) Example A

Test Case	B_1	B_2	B_3	B_4	B_5	B_6
T_1		✘	✘			✔
T_2	✘			✘		✘
T_3			✘		✘	✘
T_4	✘	✘	✘			✔

(b) Example B

A1: This algorithm implements no information. It prioritizes test cases in random order. This is the baseline method since all prioritization algorithms should have better performance than A1.

A2: This algorithm uses the historical record of failed tests to sort the tests. In Table 5 Example A, A2 will execute T_2 and T_4 randomly before T_1 and T_3 since they will reveal faults in the current test run. In Example B, A2 will execute T_2 and T_3 first, and then T_1 and T_4. We call A2 the *omniscient algorithm* since it uses information that is unavailable before prioritizing new tests. Note that if A1 represents the dumbest prioritization, then the omniscient A2 algorithm represents the best possible decisions. In the rest of this paper, we compare all results against A1 and A2 since that will let us baseline prioritization against the worse and most omniscient decisions.

B1: B1 uses the time since the last failure. A test case with less consecutive non-failure builds will be assigned with higher priority [9], [25]. In Table 5 Example A, T_4 has 0 consecutive non-failure build since it failed in B_4. Thus T_4 will be executed first. After that, T_2 has 1 consecutive failure, so it will be executed next. Moreover, T_3 and T_1 have 2 and 4 consecutive non-failure builds. Therefore, the final order in Example A is $\{T_4, T_2, T_3, T_1\}$. When same ranking scheme is applied in Example 2, it will result $\{T_3, T_2, T_1, T_4\}$. The metrics and the corresponding orders of two examples are shown in Table 6.

TABLE 6
Metrics and Orders of Algorithm B1

	Example A		Example B	
	Metric	Order	Metric	Order
T_1	4	4	2	3
T_2	1	2	1	2
T_3	2	3	0	1
T_4	0	1	2	4

B2: B2 uses the value of failure rate in metrics to prioritize test cases [1]. Failure rate is defined as:

(total number of failed builds) / (total test builds)

A test case with higher failure rate has be executed earlier. In example A, we can calculate the failure rate of T_1 to T_4 with $1/5 = 0.2$, $3/5 = 0.6$, $2/5 = 0.4$, and $3/5 = 0.6$. Thus, the algorithm will result $\{T_2, T_4, T_3, T_1\}$. In example B, the failure rates of T_1 to T_4 are $2/5 = 0.4$, $2/5 = 0.4$, $2/5 = 0.4$, $3/5 = 0.4$. We can find 3 test cases have same failure rate in example B. Thus, we will order them randomly. The metrics and the corresponding orders of two examples are shown in Table 7.

TABLE 7
Metrics and Orders of Algorithm B2

	Example A		Example B	
	Metric	Order	Metric	Order
T_1	0.2	4	0.4	2
T_2	0.6	1	0.4	4
T_3	0.4	3	0.4	3
T_4	0.6	2	0.6	1

B3: B3 implements the "Exponential Decay Metric" (mentioned earlier in this paper) to calculate the ranking values of test cases [1], [19]:

$$P_0 = B_1$$
$$P_k = \alpha B_k + (1 - \alpha)P_{k-1}, 0 \leq \alpha \leq 1, k \geq 1$$

where variables in these equations are defined as:

- B_i: The test result in build i. $B_i = 0$ if test passed and $B_i = 1$ if test failed.
- α: The learning rate. In our experiments, we test the value of α from 0 to 1 with 0.1 interval. We find $\alpha = 0.9$ reaches highest performance.
- P_j: Exponential Decay value of test case j.

A test case with higher Exponential Decay value will be executed earlier. In example A, the Exponential Decay values for T_1 to T_4 are $\{0.0001, 0.091, 0.0099, 0.9909\}$. In example B, the Exponential Decay values for T_1 to T_4 are $\{0.0099, 0.0901, 0.909, 0.01\}$. The metrics and the corresponding orders of two examples are shown in Table 8.

TABLE 8
Metrics and Orders of Algorithm B3

	Example A		Example B	
	Metric	Order	Metric	Order
T_1	0.0001	4	0.0099	4
T_2	0.091	2	0.0901	2
T_3	0.0099	3	0.909	1
T_4	0.9909	1	0.01	3

B4: B4 prioritizes test cases by implementing the ROCKET Metrics [8], [24]. In the ROCKET Metrics, prioritization value $P = \{P_1, P_2, \cdots, P_n\}$ is calculated as follow:

$$w_i = \begin{cases} 0.7, & \text{if } i = 1 \\ 0.2, & \text{if } i = 2 \\ 0.1, & \text{if } i \geq 3 \end{cases}$$

$$P_i = \sum_{j=1}^{i-1} B_j * w_{i-j}$$

where variables in this system are defined as:

- B_i: The test result in build i. $B_i = 0$ if test passed and $B_i = 1$ if test failed.
- P_j: The ROCKET value of test case j.

The prioritization value P will be ranked in descending order. Test cases will be executed in the ranked orders. In the example A in Table 5, this scheme will output the results of T_1 to T_4 with $\{0.1, 0.4, 0.2, 1.0\}$, and in the example B, ROCKET Metric of T_1 to T_4 is $\{0.2, 0.3, 0.8, 0.3\}$. We can observe that the test case has failures in the previous two builds or the test case with more failures in the past will have higher value. The metrics and the corresponding orders for two examples are shown in Table 9.

C1: The C1 algorithm was introduced by Zhu et al. in 2018. They consider the past test co-failure distributions in test case prioritization [26]. Making two test cases as a pair of tests, the co-failure score is calculated by:

$$Score(t) = prevScore(t) + (P(t = fail|t_{finished}) - 0.5)$$

TABLE 9
Metrics and Orders of Algorithm B4

	Example A		Example B	
	Metric	Order	Metric	Order
T_1	0.1	4	0.2	4
T_2	0.4	2	0.3	3
T_3	0.2	3	0.8	1
T_4	1.0	1	0.3	2

where variables in this equation are defined as:

- t: Test cases that are not executed.
- $t_{finished}$: Test case that executed just now.
- $Score(t)$: Score of test case t in current test run.
- $prevScore(t)$: Score of test case t in previous test run.

A higher score in this approach means highly correlated with the executed test cases. The test case with higher score in this approach means highly correlated with the executed test cases. In the example A in Table 5, by given T_2 failed initially, T_1 failed one time when T_2 failed in the history. Thus, the score of T_1 is updated to $0 + 1/3 - 0.5 = -0.17$. By using the same way to calculate T_3 and T_4, the scores of T_1, T_3, and T_4 will be updated from initial 0 to $\{-0.17, -0.17, 0.17\}$. Since T_4 has the highest score, T_4 is highly correlated with T_2 in this example. Therefore, T_4 will be executed next. After T_4 being executed and failed, the scores of T_1 and T_3 is updated to $\{-0.67, -0.34\}$. Since T_3 has higher score than T_1, this scheme will order T_3 before T_1. In example B, given T_3 failed, the scores of rest test cases is updated to $\{0, -0.5, 0\}$. Since T_1 and T_4 have same score, we will randomly select T_4. After T_4 is being executed and passed, the score of T_1 is updated from 0 to -0.5 since $P(t = fail|t_{finished} = pass) = 0$, and the score of T_2 is updated from -0.5 to -0.5 since $P(t = fail|t_{finished} = pass) = 0.5$. Therefore, T_2 will be executed randomly before T_1 because they have the same score. The detailed scores and corresponding orders of two examples are shown in Table 10.

TABLE 10
Metrics and Orders of Algorithm C1

	Example A		Example B	
	Metric	Order	Metric	Order
T_1	-0.17,-0.67	4	0,-0.5	4
T_2	-,-	1	-0.5,-0.5	3
T_3	-0.17,-0.34	3	-,-	1
T_4	0.17,-	2	0,-	2

C2: C2 algorithm is proposed by Cho et al. in 2016. They define that two test cases are highly correlated if their testing results change to the opposite status (flip) in two consecutive test runs [8]. Moreover, they utilize the ROCKET method to find the first failed test case. In the example A in Table 5, the ROCKET approach will locate T_4 first. After T_4 is executed and failed, we can find T_1 flips one time compare to T_4 in B_1 to B_2 because T_1 and T_4 both change their results to the opposite site. By using the same way to calculate, the flipping history for T_1 to T_3 is $\{1, 2, 2\}$. Thus, T_2 and T_3 will be randomly selected. Assume T_2 is assigned to second order, T_1 and T_3 will update their flipping history to $\{0, 1\}$. Hence T_3 will be executed before T_1. In example B, ROCKET approach will assign T_3 to the first order. The flipping history of T_1, T_2, and T_4 will update to $\{1, 2, 1\}$. After T_2 executed and failed, the flipping history will update to $\{2, 1\}$. Thus T_1 will be executed before T_4. The detailed scores and corresponding orders of two examples are shown in Table 11.

TABLE 11
Metrics and Orders of Algorithm C2

	Example A		Example B	
	Metric	Order	Metric	Order
T_1	1,0	4	1,2	3
T_2	2,-	2	2,-	2
T_3	2,1	3	-,-	1
T_4	-,-	1	1,1	4

D1: The last algorithm TERMINATOR in our experiments was proposed by Yu et al. in 2019. TERMINATOR implements an active learning based framework [10]. This approach uses execution history as features to incrementally train a support vector machine classifier. Uncertainty sampling[2] is applied until the number of detected faults exceeds some threshold N_1. After that, certainty sampling[3] is utilized until all test cases are prioritized. In the examples, we set the threshold N_1 to 2. In the example A in Table 5, with T_2 being randomly executed, and labeled as failed test case, the algorithm randomly presume T_1 as a non-relevant test case. After that, an SVM learner is trained by using the execution history of T_1 and T_2, with T_2 labeled as failed and T_1 labeled as passed. Here, we assume that the fitting results of T_1, T_3, and T_4 are $\{0.3, 0.6, 0.55\}$. Since the algorithm does not hit the threshold, uncertainty sampling is applied to this result. In this case, T_4 is selected because it is the most uncertain sample (closest to 0.5). After that, with more evidence, we assume T_1 and T_3 have prediction result $\{0.2, 0.8\}$. Since the number of failed test cases exceeds the threshold, certainty sampling will be applied. T_3 will be executed next because it has the highest predicted probability to reveal the fault. The final order in this example is $\{T_2, T_4, T_3, T_1\}$. In example B, if T_4 is randomly selected first and passed, the algorithm will continue randomly select another test case T_3. Since T_3 failed, the algorithm will start learning as example A does. The metrics and the corresponding orders for two examples are shown in Table 12.

TABLE 12
Metrics and Orders of Algorithm D1

	Example A		Example B	
	Metric	Order	Metric	Order
T_1	0.3,0.2	4	0.75,0.3	4
T_2	-,-	1	0.6,-	3
T_3	0.6,0.8	3	-,-	2
T_4	0.55,-	2	-,-	1

[2] Execute the test case with the most uncertain predicted probability.
[3] Execute the test case with the highest predicted probability.

3.5 Statistical Methods

In our study, we report the median and interquartile range (which show 50th percentile and 75th-25th percentile), of APFD results for entire test runs. We collect median and interquartile range values for each of the projects.

To make comparisons among all algorithms on a single project, we implement the Scott-Knott analysis [44]. In summary, using Scott-Knott, algorithms are sorted by their performance into some position i. Algorithms are then assigned different ranks if algorithm i's performance is significantly different to the algorithm at position $i-1$.

To be more precise Scott-Knott sorts the list of treatments (in this paper, the prioritization algorithms) by their median score. After the sorting, it then splits the list into two sub-lists. The goal for such a split is to maximize the expected value of differences in the observed performances before and after division [45]. For example, in our study, we implement 9 prioritization approaches in list l as treatments, then the possible divisions of l_1 and l_2 are $(l_1, l_2) \in \{(1,8),(2,7),(3,6),(4,5),(5,4),(6,3),(7,2),(8,1)\}$. Scott-Knott analysis then declares one of the above divisions to be the best split. The best split should maximize the difference $E(\Delta)$ in the expected mean value before and after the split:

$$E(\Delta) = \frac{|l_1|}{|l|} abs(\overline{l_1} - \overline{l})^2 + \frac{|l_2|}{|l|} abs(\overline{l_2} - \overline{l})^2 \qquad (2)$$

where:

- $|l|$, $|l_1|$, and $|l_2|$: Size of list l, l_1, and l_2.
- $\overline{l}$, $\overline{l_1}$, and $\overline{l_2}$: Mean value of list l, l_1, and l_2.

After the best split is declared by the formula above, Scott-Knott then implements some statistical hypothesis tests to check whether the division is useful or not. Here "useful" means l_1 and l_2 differ significantly by applying hypothesis test H. If the division is checked as a useful split, the Scott-Knott analysis will then run recursively on each half of the best split until no division can be made. In our study, hypothesis test H is the cliff's delta non-parametric effect size measure. Cliff's delta quantifies the number of difference between two lists of observations beyond p-values interpolation [46]. The division passes the hypothesis test if it is not a "small" effect ($Delta \geq 0.147$). The cliff's delta non-parametric effect size test explores two lists A and B with size $|A|$ and $|B|$:

$$Delta = \frac{\sum\limits_{x \in A} \sum\limits_{y \in B} \begin{cases} +1, & \text{if } x > y \\ -1, & \text{if } x < y \\ 0, & \text{if } x = y \end{cases}}{|A||B|} \qquad (3)$$

In this expression, cliff's delta estimates the probability that a value in list A is greater than a value in list B, minus the reverse probability [46]. This hypothesis test and its effect size is supported by Hess and Kromery [47].

4 Results

In this section, we will show our experimental results and answer RQs with these results. Note that RQ1 only shows results for the same closed-source project studied in the TERMINATOR paper [10] while RQ2 states the experimental results for 30 open-source projects.

TABLE 13
Scott-Knott analysis for the proprietary data from our industrial partner. In this table, "med" denotes median; blue row shows D1 algorithm, while red row shows B1/B3.

rank	what	med	IQR	
1	A1	0.50	0.02	●
2	C2	0.69	0.06	●
3	**B1**	**0.70**	**0.08**	●
3	**B3**	**0.72**	**0.08**	●
4	B2	0.74	0.08	●
4	B4	0.75	0.08	●
5	C1	0.79	0.09	●
5	**D1**	**0.80**	**0.14**	●
6	A2	0.96	0.08	●

4.1 What is the best algorithm for closed-source project? (RQ1)

To answer RQ1, we reproduce the Yu et al. study by implementing the prioritization approaches we find from the previous literature in their data set [10]. Note that, for this data, Yu et al. recommended TERMINATOR (which we call the D1 prioritization algorithm).

Table 13 shows our simulation results of 9 prioritization algorithms. We record APFD result of each test run, and calculate median value and interquartile range of APFD for all test runs. An algorithm with higher APFD value in our experiments has better performance. As described in §3.5, algorithms differ significantly if they separate in different ranks of the Scott-Knott analysis.

As seen in Table 13, as might be expected, the performance all algorithms are bounded by the dumbest A1 prioritization algorithm (which performed worse) and the omniscient A2 algorithm (that performed best).

After A2 we see that D1 and C1 are tied for the best place (in rank 5). That said, we recommend D1 over C1:

- D1 runs five times faster than C1 (in our proprietary project, 328 seconds versus 1457 seconds).
- D1 converges faster to a higher plateau of performance. Figure 2 records the percentage of failed test cases that are explored with increasing number of tests that are executed in the last test run. In Figure 2, X-axis represents the percentage of tests that are executed, and

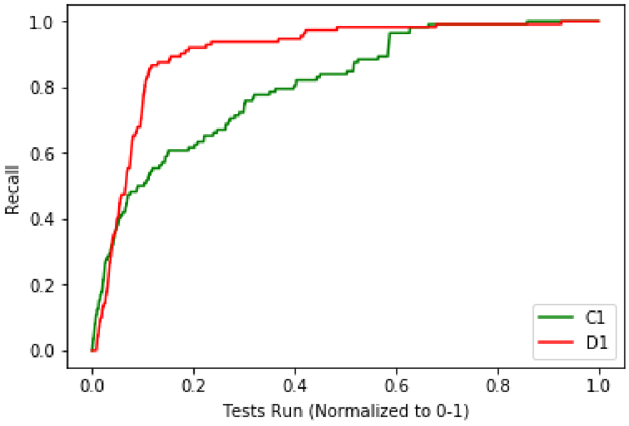

Fig. 2. Mean fault detection rates. X-axis = number of tests executed, Y-axis = "recall" (percentage of failing test suites).

Y-axis represents the percentage of failing test cases that are explored. We say *D1 converges faster* since D1 explores 85% failed test cases while C1 only explores 55% when 15% tests are executed.

Hence we answer **RQ1** as follows:

> *The* D1 *prioritization scheme, which is* TERMINATOR, *works best for that closed-source project.*

4.2 What is the best algorithm for open-source projects? (RQ2)

In order to answer RQ2, we use 10 computational science (CS) projects and 20 software engineering (SE) projects from GitHub. Table 14 shows the Scott-Knott analysis for 10 CS projects and Table 15 states the results for 20 SE projects. From comparisons among all 30 projects, we observe that for all these open-source projects, B1 and B3 always perform better than any other algorithms. Interestingly, except, algorithms B1 and B3 are ranked the same as the omniscient A2 algorithm in 8/10 of the Table 14 results and 13/21 of the Table 15 results. That is, in the majority case, B1 and B3 are performing in such a high level where they cannot be beaten.

Moreover, in our experiments, we find C1 takes a very long time in prioritizing projects which have over 800 test builds or 1500 failed test cases. For example, in the *Reaction Mechanism Generator* project, which has 850 test builds and 617 failed test cases, C1 takes around 48 hours to simulate 70% test builds. Therefore, we conclude that C1 is a very computational costly algorithm which has issues scaling up to projects with a huge number of test builds or failed test cases. C1 performs so slowly that we do not use it for our analysis of projects with more than 800 test builds or more than 1500 failed test cases.

In summary, we can conduct the answer for RQ2 based on the above results:

> *For open-source projects, the best approach is not* D1 TERMINATOR, *but rather to prioritize using either* B1, *which is* passing times since last failure *or* B3, *which implements* exponential metric.

TABLE 14

Scott-Knott analysis results for 10 open-source computational science projects. In these tables blue row denotes the performance of D1 algorithm, while red row denotes the performance of B1/B3 approach. Note that in 8/10 in these results, B1/B3 is ranked the same as the omniscient A2 method: see figures b,c,e,f,g,h,i,j.

rank	what	med	IQR
1	A1	0.51	0.45
1	**D1**	**0.54**	**0.38**
2	B2	0.79	0.46
2	C1	0.81	0.48
2	B4	0.88	0.48
2	C2	0.89	0.39
3	**B1**	**0.97**	**0.35**
3	**B3**	**0.97**	**0.32**
4	A2	0.99	0.00

(a). Project Name: parsl/parsl

rank	what	med	IQR
1	A1	0.51	0.26
1	**D1**	**0.60**	**0.36**
2	B2	0.81	0.34
2	B4	0.82	0.34
2	C2	0.84	0.40
3	C1	0.98	0.20
4	**B1**	**0.99**	**0.20**
4	**B3**	**0.99**	**0.19**
4	A2	0.99	0.15

(b). Project Name: radical-sybertools/radical

rank	what	med	IQR
1	A1	0.50	0.28
2	**D1**	**0.66**	**0.48**
3	B2	0.92	0.33
3	C2	0.94	0.28
3	B4	0.95	0.27
3	C1	0.96	0.24
4	**B1**	**0.99**	**0.16**
4	**B3**	**0.99**	**0.16**
4	A2	1.00	0.01

(c). Project Name: yt-project/yt

rank	what	med	IQR
1	A1	0.50	0.14
2	B2	0.80	0.44
2	**D1**	**0.83**	**0.34**
2	C2	0.83	0.32
2	B4	0.85	0.37
2	C1	0.89	0.39
3	**B1**	**0.98**	**0.16**
3	**B3**	**0.98**	**0.15**
4	A2	0.99	0.01

(d). Project Name: mdanalysis/mdanalysis

rank	what	med	IQR
1	A1	0.50	0.20
2	**D1**	**0.79**	**0.40**
3	C2	0.95	0.13
3	B2	0.95	0.13
3	B4	0.96	0.10
4	**B1**	**0.99**	**0.02**
4	**B3**	**0.99**	**0.01**
4	A2	1.00	0.01

(e). Project Name: unidata/metpy

rank	what	med	IQR
1	A1	0.50	0.41
1	**D1**	**0.53**	**0.48**
2	C2	0.99	0.14
2	B2	1.00	0.08
2	B4	1.00	0.05
2	**B1**	**1.00**	**0.01**
2	**B3**	**1.00**	**0.01**
2	A2	1.00	0.00

(f). Project Name: materialsproject/pymatgen

rank	what	med	IQR
1	A1	0.50	0.24
2	**D1**	**0.74**	**0.39**
3	B2	0.91	0.24
3	B4	0.92	0.23
3	C2	0.93	0.20
4	**B1**	**0.99**	**0.12**
4	**B3**	**0.99**	**0.11**
4	A2	1.00	0.01

(g). Project Name: reactionMechanism../RMG-Py

rank	what	med	IQR
1	A1	0.50	0.14
2	**D1**	**0.83**	**0.38**
2	B2	0.84	0.38
3	B4	0.92	0.30
3	C2	0.94	0.24
4	**B1**	**1.00**	**0.10**
4	**B3**	**1.00**	**0.11**
4	A2	1.00	0.00

(h). Project Name: openforcefield/openforcefield

rank	what	med	IQR
1	A1	0.50	0.29
2	**D1**	**0.69**	**0.38**
3	C2	0.99	0.06
3	B4	0.99	0.03
3	B2	0.99	0.03
3	**B1**	**1.00**	**0.01**
3	**B3**	**1.00**	**0.01**
3	A2	1.00	0.00

(i). Project Name: spotify/luigi

rank	what	med	IQR
1	A1	0.50	0.26
2	**D1**	**0.74**	**0.35**
3	C2	1.00	0.01
3	B4	1.00	0.01
3	B2	1.00	0.01
3	**B1**	**1.00**	**0.00**
3	**B3**	**1.00**	**0.00**
3	A2	1.00	0.00

(j). Project Name: galaxyProject/galaxy

TABLE 15

Scott-Knott analysis results from 20 open-source software engineering projects. In these tables, blue row marks the performance of D1 algorithm, while red row denotes the performance of B1/B3 approaches. Note that in 13/20 of these results, B1/B3 is ranked the same as the omniscient A2 method: see figures b,c,d,e,f,g,h,i,m,o,q,s,t. Algorithms with n/a mean they are too expensive to finish so that they are in the lowest rank.

rank	what	med	IQR
1	A1	0.52	0.28
1	**D1**	**0.59**	**0.42**
2	C2	0.89	0.32
2	C1	0.91	0.18
2	B2	0.91	0.23
2	B4	0.95	0.22
3	**B1**	**0.97**	**0.17**
3	**B3**	**0.97**	**0.14**
4	A2	0.99	0.02

(a). Project Name: loomio/loomio

rank	what	med	IQR
1	A1	0.48	0.31
2	**D1**	**0.65**	**0.32**
3	C2	0.75	0.25
4	C1	0.94	0.14
4	B2	0.95	0.20
5	B4	0.97	0.16
5	**B1**	**0.98**	**0.09**
5	**B3**	**0.98**	**0.06**
5	A2	0.98	0.00

(b). Project Name: languagetool-org/languagetool

rank	what	med	IQR
1	A1	0.49	0.23
2	**D1**	**0.67**	**0.37**
3	C2	0.78	0.28
4	B2	0.92	0.19
4	C1	0.93	0.14
4	B4	0.94	0.14
5	**B1**	**0.98**	**0.04**
5	**B3**	**0.98**	**0.03**
5	A2	0.99	0.01

(c). Project Name: deeplearning4j/deeplearning4j

rank	what	med	IQR
1	A1	0.51	0.23
2	**D1**	**0.67**	**0.26**
3	C2	0.78	0.22
4	C1	0.92	0.10
4	B2	0.93	0.10
4	B4	0.94	0.09
5	**B1**	**0.96**	**0.04**
5	**B3**	**0.96**	**0.03**
5	A2	0.97	0.01

(d). Project Name: Unidata/thredds

rank	what	med	IQR
1	A1	0.49	0.24
2	**D1**	**0.68**	**0.37**
2	C2	0.68	0.36
3	B2	0.96	0.16
3	C1	0.97	0.09
3	B4	0.98	0.12
4	**B1**	**0.99**	**0.01**
4	**B3**	**0.99**	**0.02**
4	A2	1.00	0.01

(e). Project Name: nutzam/nutz

rank	what	med	IQR
1	A1	0.50	0.22
2	**D1**	**0.73**	**0.35**
3	C2	0.78	0.24
4	C1	0.97	0.04
4	B2	0.97	0.09
4	B4	0.97	0.07
5	**B1**	**0.99**	**0.02**
5	**B3**	**0.99**	**0.02**
5	A2	0.99	0.00

(f). Project Name: structr/structr

rank	what	med	IQR
1	A1	0.47	0.24
2	**D1**	**0.63**	**0.34**
3	C2	0.72	0.24
4	B2	0.94	0.24
4	C1	0.95	0.17
4	B4	0.96	0.24
5	**B1**	**0.98**	**0.13**
5	**B3**	**0.98**	**0.10**
5	A2	0.99	0.01

(g). Project Name: ocpsoft/rewrite

rank	what	med	IQR
1	A1	0.51	0.23
2	**D1**	**0.74**	**0.29**
3	C2	0.80	0.20
4	C1	0.98	0.02
4	B4	0.98	0.03
4	B2	0.98	0.04
4	**B1**	**0.98**	**0.07**
4	**B3**	**0.98**	**0.06**
4	A2	0.99	0.00

(h). Project Name: eclipse/jetty.project

rank	what	med	IQR
1	A1	0.51	0.28
2	**D1**	**0.63**	**0.36**
3	C2	0.72	0.23
4	C1	0.96	0.03
4	B2	0.97	0.04
4	B4	0.98	0.03
4	**B1**	**0.98**	**0.01**
4	**B3**	**0.98**	**0.00**
4	A2	0.98	0.00

(i): Project Name: square/okhttp

rank	what	med	IQR
1	A1	0.53	0.34
1	**D1**	**0.59**	**0.46**
2	B2	0.88	0.22
2	C2	0.88	0.36
2	B4	0.89	0.28
2	C1	0.90	0.15
3	**B1**	**0.96**	**0.15**
3	**B3**	**0.96**	**0.15**
4	A2	0.99	0.00

(j). Project Name: openSUSE/open-build-service

rank	what	med	IQR
1	A1	0.49	0.19
2	**D1**	**0.63**	**0.32**
3	B2	0.71	0.31
3	B4	0.72	0.30
3	C1	0.72	0.34
3	C2	0.73	0.19
4	**B1**	**0.93**	**0.26**
4	**B3**	**0.94**	**0.27**
5	A2	0.99	0.2

(k). Project Name: thinkaurelius/titan

rank	what	med	IQR
1	A1	0.50	0.24
2	**D1**	**0.69**	**0.37**
3	C2	0.70	0.26
4	C1	0.91	0.21
4	B2	0.91	0.24
4	B4	0.92	0.23
5	**B1**	**0.96**	**0.12**
5	**B3**	**0.96**	**0.12**
6	A2	0.99	0.03

(l): Project Name: Graylog2/graylog2-server

rank	what	med	IQR
1	A1	0.48	0.32
2	**D1**	**0.66**	**0.36**
3	C2	0.95	0.28
4	B4	0.98	0.03
4	**B1**	**0.98**	**0.03**
4	B2	0.98	0.03
4	**B3**	**0.98**	**0.03**
4	C1	0.98	0.03
4	A2	0.98	0.01

(m). Project Name: puppetlabs/puppet

rank	what	med	IQR
1	A1	0.51	0.31
2	**D1**	**0.64**	**0.41**
3	B2	0.83	0.28
3	C2	0.84	0.24
3	B4	0.85	0.24
3	C1	0.88	0.25
4	**B1**	**0.98**	**0.09**
4	**B3**	**0.98**	**0.09**
5	A2	1.00	0.02

(n). Project Name: middleman/middleman

rank	what	med	IQR
1	A1	0.48	0.45
1	**D1**	**0.51**	**0.38**
2	B2	0.96	0.33
2	C1	0.96	0.15
2	B4	0.97	0.29
2	C2	0.97	0.30
3	**B1**	**0.99**	**0.03**
3	**B3**	**0.99**	**0.03**
3	A2	0.99	0.00

(o): Project Name: locomotivecms/engine

rank	what	med	IQR
1	A1	0.50	0.24
2	**D1**	**0.72**	**0.39**
3	B2	0.78	0.36
3	B4	0.79	0.37
3	C2	0.81	0.33
3	C1	0.83	0.32
4	**B1**	**0.90**	**0.17**
4	**B3**	**0.94**	**0.15**
5	A2	0.99	0.08

(p). Project Name: diaspora/diaspora

rank	what	med	IQR
1	A1	0.50	0.25
2	**D1**	**0.67**	**0.41**
3	C2	0.75	0.24
4	B2	0.97	0.10
4	B4	0.97	0.10
5	C1	1.00	0.03
5	**B1**	**1.00**	**0.01**
5	**B3**	**1.00**	**0.01**
5	A2	1.00	0.00

(q). Project Name: facebook/presto

rank	what	med	IQR
1	A1	0.52	0.23
2	**D1**	**0.70**	**0.28**
3	C2	0.73	0.32
4	C1	0.82	0.16
4	B2	0.83	0.17
4	B4	0.83	0.17
5	**B1**	**0.86**	**0.19**
5	**B3**	**0.86**	**0.17**
6	A2	0.98	0.15

(r): Project Name: rspec/rspec-core

rank	what	med	IQR
1	A1	0.50	0.12
1	C1	n/a	n/a
2	C2	0.86	0.11
2	**D1**	**0.88**	**0.17**
2	B2	0.88	0.14
2	B4	0.88	0.15
3	**B1**	**0.98**	**0.02**
3	**B3**	**0.98**	**0.02**
3	A2	0.99	0.02

(s). Project Name: rails/rails

rank	what	med	IQR
1	A1	0.50	0.04
1	**D1**	**n/a**	**n/a**
1	C1	n/a	n/a
1	C2	n/a	n/a
2	B2	0.97	0.07
2	B4	0.97	0.07
3	**B1**	**0.99**	**0.04**
3	**B3**	**0.99**	**0.04**
3	A2	0.99	0.01

(t). Project Name: jruby/jruby

4.3 Are different prioritization algorithms perform various in the open-source projects and the closed-source project? (RQ3)

To answer RQ3, we look at the B1/B3 and D1 results in Table 13, Table 14, and Table 15. We highlight D1 result with blue and the B1/B3 results with red. Note that the ranking of these algorithms is reversed for our closed-source and open-source examples:

- As shown in Table 13, for our close-sourced case study, D1 was seen to perform much better than B1/B3.
- However, as shown in Table 14 and Table 15, for open-source projects, that ranking is completely reverse,

Based on these points, we can answer RQ3 that

> *Test case prioritization schemes that work best for the industrial closed-source project can work worse for open-source projects (and vice versa)*

5 Discussion

5.1 Performance of purposed Prioritization Algorithms in Different Sources of Projects

In our study, we conduct that D1 performs "best" in the industrial closed-source project, but "worst" in open-source projects. On the opposite, B1 and B3 have the best performance in open-source projects, but worse in the industrial closed-source project. How to explain this difference?

TABLE 16
Statistics on the number of failure in each project. The first row shown in blue shows our closed-source data while the other rows come from open-sources projects. Rows are sorted by the meduan valyes

	failed cases per build				
Project name	10%	30%	50%	70%	90%
Proprietary data	90	112	143	319	791
rails	1	3	11	19	24
Mdanalysis	1	3	10	13	66
openforcefield	1	2	7	21	96
Eclipse	2	3	3	3	6
ocpsoft	1	2	3	4	8
reaction...	1	1	3	5	18
unidata@metpy	1	2	3	8	13
Unidata@thredds	2	3	3	4	6
Deeplearning4j	2	2	2	3	6
Diaspora	1	1	2	4	30
Facebook	2	2	2	2	3
Galaxyproject	2	2	2	2	5
Graylog2	2	2	2	4	5
Languagetool-org	2	2	2	2	4
nutzam	2	2	2	3	6
puppetlabs	1	1	2	2	4
rspec	1	1	2	18	22
spotify	1	2	2	3	4
square	2	2	2	2	2
structr	2	2	2	4	7
thinkaurelius	2	2	2	6	54
yt-project	1	1	2	4	15
Locomotivecms	1	1	1	1	2
Materialsproject	1	1	1	1	2
Middleman	1	1	1	3	14
openSUSE	1	1	1	2	15
parsl	1	1	1	1	2
radical-sybertools	1	1	1	4	45
loomio	1	1	1	3	14

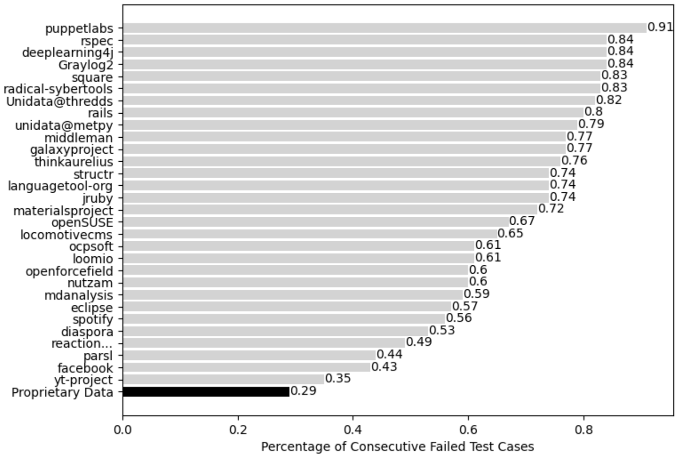

Fig. 3. Percentage of consecutive failed test cases. Our proprietary closed-source project is shown last. .

To answer this question, we first recall the internal details of our different algorithms:

- The D1 algorithm learns from the results of other test cases in the current build.
- On the other hand, B1 and B3 are much more straight forward, since they only need the previous test results.

From the above, we would predict that D1 works best when there are more failed test cases per build in the proprietary project than our open-source projects. To check this conjecture, we went back to count the number of **failed test cases per build**. Since there are many such builds, we say need to look at the distribution across the entire project. Accordingly we report the 10th, 30th, 50th, 70th, 90th percentiles of those distributions. We say that our conjecture is supported if the number of failed test cases within a build is (a) markedly different between open-source and proprietary and (b) much larger in proprietary.

Table 16 shows those results. As predicted by our conjecture, we see that the max number of failed tests per build is markedly different and larger in our proprietary than otherwise. For example, with only one exception (MdAnalysis), the 90th percentile values from open-source projects are smaller than the 10th percentile value of our proprietary project.

This observation leads to the following statement about when to select what test case prioritzation methods:

- If projects do frequent builds that address a small number of bugs each time, then there is little information in each build. In this case, we need to look more into historical data (e.g. B1 and B3).
- On the other hand, if a team tends towards 'big bang" engineering where a release fixes multiple test failures (and, most likely, releases take longer to be generated), then there is much information in each build. In this case, an active learning approach (e.g. D1) can achieve much by reflecting over all the data in the current build.

To further support our last point, we make the following observation. The premise of the last point is that for "big-bang" projects, more information can be obtained by not only analyze prior ones, but also the current build. To test that premise, we compare how much previous failing build information is available from the historical record for our proprietary and open-source projects.

Figure 3 shows the percent of times a failed test (from the current build) also fails in the previous build. As can be seen in that figure, in open source projects, 75% (median) of the failing tests in the current build also failed in the last build. However, in our proprietary project, the majority (nearly three quarters) of failures in this build did **not** appear in the last build. Hence we can explain now why B1 and B3 work better in open-source than proprietary project:

- B1 and B3 reflect more in the history of a project.
- And in that history, open-source projects have more previous build failure information.

Overall, the picture we are offering here is that open-source projects are more earnest about removing bugs as soon as possible. Hence:

- Test case prioritization methods for proprietary projects should make most use of *the tests from current build as well as their connection to the execution history*.
- While test case prioritization methods for open-source projects should only make most use of *the past results*.

5.2 Efficiency of Prioritization Algorithms

To conclude this study, we offer a brief note on the efficiency of the different prioritization algorithms. Efficiency can be an important component in judging whether a prioritization scheme is useful or not. An algorithm can be regarded worse than others if it takes a long duration to prioritize test cases.

In Table 17, we list simulation time for each of the selected algorithm. In this table, **n/a** means the algorithm takes a very long time (over 48 hours) in simulation, so we will not consider it even though its APFD is very high.

From Table 17, we find that our proposed algorithms B1 and B3 for open-source projects have very short execution time (marked in light gray). The reason they are fast is that they only need to analyze execution history one time for each test case (which is an $\mathcal{O}(n)$ analysis). Since most of the open-source projects have very large builds and lots of test cases, this finding consolidate our conclusion that B1 and B3 are the best prioritization algorithms in open-source projects since:

- B1 and B3 have the best performance in prioritizing open-source projects.
- B1 and B3 have fast simulation speed in prioritizing large open-source projects.

That said, despite their efficiency, B1 and B3 are not applicable in our industrial closed-source project:

- In Table 17, we can observe that D1 can finish test case prioritization in 5 minutes with outstanding performance, which is acceptable (marked in dark gray). Although B1 and B3 only take a few seconds to finish the same task, we still prefer D1 since it can increase fault detection rates significantly.
- Also, from Table 13, we can find B1 and B3 are only in the rank 3. D1 has a much better performance than B1 and B3.

By taking the above reasons together, we conclude that B1 and B3 are not applicable in the closed-source project even though they have the shortest simulation time.

6 Threats to Validity

This section discusses issues raised by Feldt et al. [48]

Conclusion validity: Different treatments to simulation results may cause various conclusions. In our experiments, we implement Scott-Knott analysis to the APFD results of test runs. Prioritization algorithms differ significantly if they distribute in different ranks.

Metric validity: We implement the weighted average of the percentage of faults detected (APFD) to evaluate the performance of prioritization approaches. This evaluation metric assumes all test cases have the same cost and the same severity. However, some test cases may take longer to execute than others. This may be a threat to evaluation validity. In our future work, we plan to collect the cost of test cases so that we can implement a better evaluation metric called the average percentage of faults detected with cost (APFDc) [28].

Sampling validity: One way to characterize this paper is as a response to the sampling bias problem in previous work. As said in the introduction, Yu et al. [10] reported that the D1 TERMINATOR test case prioritization algorithm was better than dozens of alternatives. However, TERMINATOR

TABLE 17
Run time for all algorithms (unit: (s)). Dark gray marks the performance of D1 in the proprietary closed-source project from our industrial partner, which is an acceptable run time. Light gray marks the performances of B1 and B3 in the open-source projects, which are much shorter than D1.

Project Name	B1	B2	B3	B4	C1	C2	D1	Project Name	B1	B2	B3	B4	C1	C2	D1
Unidata/thredds	0.2	0.3	0.3	0.4	8.4	1.2	1.7	Middleman	3.4	6.2	8.0	9.4	598.8	34.5	33.0
OpenSUSE	0.2	0.3	0.3	0.3	11.4	1.1	1.9	Diaspora	9.1	20.6	26.4	31.3	1738.0	255.8	78.3
Thinkaurelius	0.2	0.3	0.3	0.4	16.8	2.7	2.9	Structr	14.4	19.4	25.1	31.5	3309.9	129.9	147.9
Loomio	0.2	0.4	0.5	0.6	21.9	3.0	3.1	Yt-project	14.7	27.8	34.6	39.9	12494.4	315.8	563.4
Languagetool...	0.3	0.4	0.5	0.6	17.3	1.1	2.2	Mdanalysis	24.5	38.1	47.8	55.5	34336.1	1271.9	3542.5
ocpsoft	0.2	0.3	0.3	0.4	19.3	1.2	3.3	Facebook	43.0	45.7	56.4	62.7	67635.8	281.7	4646.1
Locomotivems	0.2	0.3	0.3	0.4	8.4	1.2	2.2	Reaction..	45.0	70.4	86.3	105.5	n/a	924.2	2551.4
Parsl	0.3	0.5	0.7	0.8	27.4	1.2	3.7	Openforcefield	49.2	81.8	102.4	117.6	n/a	4685.4	33962.7
Graylog2	0.4	0.4	0.5	0.6	22.3	1.7	3.2	Unidata	105.7	158.5	196.3	244.9	n/a	1793.8	4271.5
Eclipse	0.4	0.6	0.8	1.0	86.1	4.0	8.3	Materials..	167.3	183.1	221.2	277.0	n/a	873.5	6439.4
Rspec	0.5	0.7	0.8	1.0	29.8	7.2	4.0	Spotify	564.5	959.8	1203.1	1541.1	n/a	5525.3	15961.1
Radical-syber..	0.9	1.3	1.7	1.9	69.1	9.7	6.2	Rails	2950.1	5376.4	6716.6	8339.4	n/a	68648.1	82601.2
Deeplearning4j	1.0	1.3	1.6	1.9	99.0	7.4	8.8	Galaxy..	9721.8	9169.9	11297.2	14466.7	n/a	56929.1	205651.6
Puppetlabs	1.3	1.4	1.6	1.8	54.1	5.7	5.3	Jruby	1081.7	1797.7	2393.1	2946.3	n/a	n/a	n/a
Nutzam	2.2	3.4	4.5	5.4	792.7	17.9	57.4								
Square	2.7	3.1	3.9	4.7	168.5	11.2	13.1	Proprietary data	1.7	2.3	2.2	2.4	1457.5	285.9	327.7

was developed for the closed-source proprietary software. This raises the question explored here: does TERMINATOR work for other kinds of projects (e.g. open-source projects)? While this paper mitigates some of the sampling bias seen in Yu et al., it is also true that other data, not used in this study, could invalidate our results. Sampling bias threatens any paper on analytics (not just this one) since conclusions that hold for one project may not hold for any other. No paper can explore all data sets – the best we can do (and we have done) is carefully documenting our methods and placing our tools on a repository that others can access (so the community can easily apply our methods to their data).

More specifically, this study reports results from dozens of open-source projects. While, ideally, we should also have report on an equal number of closed-source projects, industrial SE research does not work that way. Like many other researchers, we have spend years carefully nurturing our industrial contacts (and reporting on the results of those interactions [49]–[58]). Also, like many other researchers, we find it hard to get data released from industrial clients. Hence, as shown in Table 1, researchers in this area have only been able to use data from 0, 1 or 2 closed-source projects.

For us, the only closed-source data we can report here comes from the Yu et al.'s TERMINATOR study. Since such closed-source data is so scarce, we take care to make the best use of it:

- §4.3 showed that there was an unequivocal difference in results from the closed-source TERMINATOR study and our 30 open-source projects. Specifically, in all 30 open-source projects, the methods learned by TERMINATOR (learned from closed-source projects) failed very badly.
- In §5.1, we showed that that difference can be explained due to fundamental differences in the nature of open and closed-source projects (open-source developers try to fix less bugs in consecutive builds than close sourced projects).

The lessons stated in our conclusion section are based on these two observations.

7 Conclusion and Future Work

Regression testing is an important component in software testing and development. Better prioritization schemes can help developers detect more faults within a limited time. Therefore, test case prioritization is widely studied in the software testing region.

By searching the literature, we found nine prioritization schemes that prioritize test cases by utilizing the information of execution history. These were applied to one closed-source project and 30 open-source projects. The differences in results between our closed-source project and the open-source projects was clear:

- The D1 TERMINATOR algorithm performs the best in the industrial closed-source project, but performs the worst in open-source projects.
- Further, algorithm B1 and B3 has the highest performance in open-source projects, while they are worse than D1 in our closed-source project.

§4.3 of this paper argued that this difference was fundamental to the nature of open and closed-source projects; specifically:

- Open-source developers try to fix less bugs in consecutive builds than close sourced projects;
- This has implications on how much can be learned from one build;
- This, in turn, has implications on what prioritization method works best.

Table 1 of this report shows that this study uses far more projects than anything listed there within the last decade. Nevertheless, like many other researchers, we only have limited access to closed-source projects. Hence, we take care to express our conclusions appropriately. When answering **RQ3**, when recommending better prioritization schemes, we take care to say "can work worse" rather than "will always work' Also, we express our general conclusion not in terms of open-vs-closed but rather in terms of the need to tuning prioritization methods to the projects at hand

Specifically, the general lesson we offer is:

> *It is ill-advised to always apply one prioritization scheme to all projects since prioritization requires tuning to different projects types.*

As to future work, we suggest the following. It is no longer enough to report "the" best prioritization scheme. Research in this area should pivot to a related question; i.e. how can we, in a cost and time effective manner, explore different test case prioritization for the current data. Hence we say that future work should:

- Collect the cost of test cases from more open-source projects (to find better performance evaluation metrics).
- Make more comparisons by implementing more prioritization algorithms for both open-source projects and closed-source projects.
- Collect more projects from different sources to verify our findings in both open-source projects and closed-source projects.
- Seek patterns of how APFD score changes when test run increases for each algorithm in the large projects. By implementing feature extraction techniques and machine learners, we can try to predict the best test case prioritization algorithm for the projects.
- Develop a prioritization scheme that can work well for in both open-source projects and closed-source projects.

We predict that the last task would be particularly hard to complete.

Acknowledgements

This work was partially funded by a research grant from the National Science Foundation (CCF #1703487).

References

[1] Y. Fazlalizadeh, A. Khalilian, M. A. Azgomi, and S. Parsa, "Prioritizing test cases for resource constraint environments using historical test case performance data," in *2009 2nd IEEE International Conference on Computer Science and Information Technology*. IEEE, 2009, pp. 190–195.

[2] Y. Lu, Y. Lou, S. Cheng, L. Zhang, D. Hao, Y. Zhou, and L. Zhang, "How does regression test prioritization perform in real-world software evolution?" in *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, 2016, pp. 535–546.

[3] S. Mahajan, S. D. Joshi, and V. Khanaa, "Component-based software system test case prioritization with genetic algorithm decoding technique using java platform," in *2015 International Conference on Computing Communication Control and Automation*, 2015, pp. 847–851.

[4] P. K. Chittimalli and M. J. Harrold, "Re-computing coverage information to assist regression testing," in *2007 IEEE International Conference on Software Maintenance*, 2007, pp. 164–173.

[5] M. Beller, G. Gousios, A. Panichella, S. Proksch, S. Amann, and A. Zaidman, "Developer testing in the ide: Patterns, beliefs, and behavior," *IEEE Transactions on Software Engineering*, vol. 45, no. 3, pp. 261–284, 2019.

[6] M. Beller, G. Gousios, A. Panichella, and A. Zaidman, "When, how, and why developers (do not) test in their ides," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, 2015, pp. 179–190.

[7] W. E. Wong, J. R. Horgan, S. London, and H. Agrawal, "A study of effective regression testing in practice," in *Proceedings The Eighth International Symposium on Software Reliability Engineering*, 1997, pp. 264–274.

[8] Y. Cho, J. Kim, and E. Lee, "History-based test case prioritization for failure information," in *2016 23rd Asia-Pacific Software Engineering Conference (APSEC)*. IEEE, 2016, pp. 385–388.

[9] S. Elbaum, G. Rothermel, and J. Penix, "Techniques for improving regression testing in continuous integration development environments," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2014, pp. 235–245.

[10] Z. Yu, F. Fahid, T. Menzies, G. Rothermel, K. Patrick, and S. Cherian, "Terminator: better automated ui test case prioritization," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019, pp. 883–894.

[11] S. Koch and G. Schneider, "Effort, co-operation and co-ordination in an open source software project: Gnome," *Information Systems Journal*, vol. 12, no. 1, pp. 27–42, 2002.

[12] U. Raja and M. J. Tretter, "Defining and evaluating a measure of open source project survivability," *IEEE Transactions on Software Engineering*, vol. 38, no. 1, pp. 163–174, 2012.

[13] S. Saltis, "Comparing open source software vs closed source software," *Core DNA*, 2018.

[14] M. Beller, G. Gousios, and A. Zaidman, "Travistorrent: Synthesizing travis ci and github for full-stack research on continuous integration," in *Proceedings of the 14th working conference on mining software repositories*, 2017.

[15] ——, "Oops, my tests broke the build: An explorative analysis of travis ci with github," in *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. IEEE, 2017, pp. 356–367.

[16] G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold, "Prioritizing test cases for regression testing," *IEEE Transactions on software engineering*, vol. 27, no. 10, pp. 929–948, 2001.

[17] S. Elbaum, A. G. Malishevsky, and G. Rothermel, "Test case prioritization: A family of empirical studies," *IEEE transactions on software engineering*, vol. 28, no. 2, pp. 159–182, 2002.

[18] Z. Li, M. Harman, and R. M. Hierons, "Search algorithms for regression test case prioritization," *IEEE Transactions on software engineering*, vol. 33, no. 4, pp. 225–237, 2007.

[19] J.-M. Kim and A. Porter, "A history-based test prioritization technique for regression testing in resource constrained environments," in *Proceedings of the 24th international conference on software engineering*, 2002, pp. 119–129.

[20] B. Jiang, Z. Zhang, W. K. Chan, and T. Tse, "Adaptive random test case prioritization," in *2009 IEEE/ACM International Conference on Automated Software Engineering*. IEEE, 2009, pp. 233–244.

[21] H. Srikanth, L. Williams, and J. Osborne, "System test case prioritization of new and regression test cases," in *2005 International Symposium on Empirical Software Engineering, 2005.* IEEE, 2005, pp. 10–pp.

[22] R. Carlson, H. Do, and A. Denton, "A clustering approach to improving test case prioritization: An industrial case study," in *2011 27th IEEE International Conference on Software Maintenance (ICSM)*. IEEE, 2011, pp. 382–391.

[23] B. Qu, C. Nie, B. Xu, and X. Zhang, "Test case prioritization for black box testing," in *31st Annual International Computer Software and Applications Conference (COMPSAC 2007)*, vol. 1. IEEE, 2007, pp. 465–474.

[24] D. Marijan, A. Gotlieb, and S. Sen, "Test case prioritization for continuous regression testing: An industrial case study," in *2013 IEEE International Conference on Software Maintenance*, 2013, pp. 540–543.

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

[26] Y. Zhu, E. Shihab, and P. C. Rigby, "Test re-prioritization in continuous testing environments," in *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2018, pp. 69–79.

[27] G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold, "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, 1999, pp. 179–188.

[28] S. Elbaum, A. Malishevsky, and G. Rothermel, "Incorporating varying test costs and fault severities into test case prioritization," in *Proceedings of the 23rd International Conference on Software Engineering. ICSE 2001*. IEEE, 2001, pp. 329–338.

[29] H. Do and G. Rothermel, "On the use of mutation faults in empirical assessments of test case prioritization techniques," *IEEE Transactions on Software Engineering*, vol. 32, no. 9, pp. 733–752, 2006.

[30] S. Haidry and T. Miller, "Using dependency structures for prioritization of functional test suites," *IEEE Transactions on Software Engineering*, vol. 39, no. 2, pp. 258–275, 2013.

[31] C. Malz, N. Jazdi, and P. Gohner, "Prioritization of test cases using software agents and fuzzy logic," in *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*, 2012, pp. 483–486.

[32] J. Zemlin, "If you can't measure it, you can't improve it. https://www. linux.com/news/if-you-cant-measure-it-you-cant-improve-it-chaoss-project-creates- tools-analyze-software/," 2017.

[33] A. Vance, "Legal sites plan revamps as rivals undercut price," *The New York Times*, 2010.

[34] S. Elbaum, G. Rothermel, S. Kanduri, and A. G. Malishevsky, "Selecting a cost-effective test case prioritization technique," *Software Quality Journal*, vol. 12, no. 3, pp. 185–210, 2004.

[35] A. G. Malishevsky, J. R. Ruthruff, G. Rothermel, and S. Elbaum, "Cost-cognizant test case prioritization," Technical Report TR-UNL-CSE-2006-0004, University of Nebraska-Lincoln, Tech. Rep., 2006.

[36] L. Zhang, S.-S. Hou, C. Guo, T. Xie, and H. Mei, "Time-aware test-case prioritization using integer linear programming," in *Proceedings of the eighteenth international symposium on Software testing and analysis*, 2009, pp. 213–224.

[37] D. Jeffrey and N. Gupta, "Test case prioritization using relevant slices," in *30th Annual International Computer Software and Applications Conference (COMPSAC'06)*, vol. 1. IEEE, 2006, pp. 411–420.

[38] H. Tu, R. Agrawal, and T. Menzies, "The changing nature of computational science software," *arXiv preprint arXiv:2003.05922*, 2020.

[39] E. Kalliamvakou, G. Gousios, K. Blincoe, L. Singer, D. M. German, and D. Damian, "The promises and perils of mining github," in *Proceedings of the 11th working conference on mining software repositories*, 2014, pp. 92–101.

[40] N. Munaiah, S. Kroh, C. Cabrey, and M. Nagappan, "Curating github for engineered software projects," *Empirical Software Engineering*, vol. 22, no. 6, pp. 3219–3253, 2017.

[41] "Company news; a name change is planned for mead data central," *The New York Times.*, 1994. [Online]. Available: https://www.nytimes.com/1994/12/02/business/company-news-a-name-change-is-planned-for-mead-data-central.html?src=pm

[42] A. Vance, "Legal sites plan revamps as rivals undercut price," *The New York Times.*, 2010. [Online]. Available: https://www.nytimes.com/2010/01/25/technology/25westlaw.html?_r=1&ref=reedelsevier

[43] S. Miller, "For future reference, a pioneer in online reading," *The Wall Street Journal*, 2012. [Online]. Available: https://www.wsj.com/articles/SB10001424052970203721704577157211501855648?KEYWORDS=lexisnexis

[44] N. Mittas and L. Angelis, "Ranking and clustering software cost estimation models through a multiple comparisons algorithm,"

IEEE Transactions on software engineering, vol. 39, no. 4, pp. 537–551, 2012.

[45] T. Xia, R. Krishna, J. Chen, G. Mathew, X. Shen, and T. Menzies, “Hyperparameter optimization for effort estimation,” *arXiv preprint arXiv:1805.00336*, 2018.

[46] G. Macbeth, E. Razumiejczyk, and R. D. Ledesma, “Cliff’s delta calculator: A non-parametric effect size program for two groups of observations,” *Universitas Psychologica*, vol. 10, no. 2, pp. 545–555, 2011.

[47] M. R. Hess and J. D. Kromrey, “Robust confidence intervals for effect sizes: A comparative study of cohen’sd and cliff’s delta under non-normality and heterogeneous variances,” in *annual meeting of the American Educational Research Association*, 2004, pp. 1–30.

[48] R. Feldt and A. Magazinius, “Validity threats in empirical software engineering research-an initial survey.” in *Seke*, 2010, pp. 374–379.

[49] N. Shrikanth and T. Menzies, “Assessing practitioner beliefs,” *arXiv preprint arXiv:1912.10093*, 2019.

[50] J. Chen, J. Chakraborty, P. Clark, K. Haverlock, S. Cherian, and T. Menzies, “Predicting breakdowns in cloud services (with spike),” in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019, pp. 916–924.

[51] J. Wang, S. Wang, J. Chen, T. Menzies, Q. Cui, M. Xie, and Q. Wang, “Characterizing crowds to better optimize worker recommendation in crowdsourced testing,” *IEEE Transactions on Software Engineering*, 2019.

[52] A. Agrawal, A. Rahman, R. Krishna, A. Sobran, and T. Menzies, “We don’t need another hero? the impact of” heroes” on software development,” in *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice*, 2018, pp. 245–253.

[53] R. Krishna, A. Agrawal, A. Rahman, A. Sobran, and T. Menzies, “What is the connection between issues, bugs, and enhancements?” in *2018 IEEE/ACM 40th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*. IEEE, 2018, pp. 306–315.

[54] J. Hihn, M. Saing, E. Huntington, J. Johnson, T. Menzies, and G. Mathew, “The nasa analogy software cost model: A web-based cost analysis tool,” in *2017 IEEE Aerospace Conference*. IEEE, 2017, pp. 1–17.

[55] L. Layman, A. P. Nikora, J. Meek, and T. Menzies, “Topic modeling of nasa space system problem reports: research in practice,” in *Proceedings of the 13th International Conference on Mining Software Repositories*, 2016, pp. 303–314.

[56] T. Menzies, W. Nichols, F. Shull, and L. Layman, “Are delayed issues harder to resolve? revisiting cost-to-fix of defects throughout the lifecycle,” *Empirical Software Engineering*, vol. 22, no. 4, pp. 1903–1935, 2017.

[57] E. Kocaguneli, T. Zimmermann, C. Bird, N. Nagappan, and T. Menzies, “Distributed development considered harmful?” in *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 2013, pp. 882–890.

[58] T. Menzies, M. Benson, K. Costello, C. Moats, M. Northey, and J. Richardson, “Learning better iv&v practices,” *Innovations in Systems and Software Engineering*, vol. 4, no. 2, pp. 169–183, 2008.

Xiao Ling is a first-year PhD student in Computer Science at NC State University. His research interests include automated software testing and machine learning for software engineering.

Rishabh Agrawal is a masters student in computer science department at NC State University. His research interests include machine learning for software engineering, data mining and deep learning.

Tim Menzies (IEEE Fellow, Ph.D. UNSW, 1995) is a Professor in computer science at NC State University, USA, where he teaches software engineering, automated software engineering, and programming languages. His research interests include software engineering (SE), data mining, artificial intelligence, and search-based SE, open access science. For more information, please visit http://menzies.us.