Reinforcement Learning for Test Case Prioritization

Mojtaba Bagherzadeh, Nafiseh Kahani, and Lionel Briand, *Fellow, IEEE*

Abstract—Continuous Integration (CI) significantly reduces integration problems, speeds up development time, and shortens release time. However, it also introduces new challenges for quality assurance activities, including regression testing, which is the focus of this work. Though various approaches for test case prioritization have shown to be very promising in the context of regression testing, specific techniques must be designed to deal with the dynamic nature and timing constraints of CI. Recently, Reinforcement Learning (RL) has shown great potential in various challenging scenarios that require continuous adaptation, such as game playing, real-time ads bidding, and recommender systems. Inspired by this line of work and building on initial efforts in supporting test case prioritization with RL techniques, we perform here a comprehensive investigation of RL-based test case prioritization in a CI context. To this end, taking test case prioritization as a ranking problem, we model the sequential interactions between the CI environment and a test case prioritization agent as an RL problem, using three alternative ranking models. We then rely on carefully selected and tailored state-of-the-art RL techniques to automatically and continuously learn a test case prioritization strategy, whose objective is to be as close as possible to the optimal one. Our extensive experimental analysis shows that the best RL solutions provide a significant accuracy improvement over previous RL-based work, with prioritization strategies getting close to being optimal, thus paving the way for using RL to prioritize test cases in a CI context.

Index Terms—Continuous integration, CI, reinforcement learning, test prioritization

✦

1 INTRODUCTION

Following the common practice of Continuous Integration (CI), software developers integrate their work more frequently with the mainline code base, often several times a day [1]. Overall, CI significantly reduces integration problems, speeds up development time, and shortens release time [2]. However, it also introduces new challenges regarding quality assurance activities. Regression testing is most particularly affected due to (1) a very dynamic environment resulting from frequent changes in source code and test cases, (2) timing constraints, since regression testing should be fast enough to enable the code to be built and tested frequently.

By default, regression testing runs all previously executed test cases to gain confidence that new changes do not break existing functionality (run-them-all approach). However, depending on the size of the code base, the number of test cases can be huge and their execution often requires many servers and can take hours or even days to complete. Test case selection and prioritization techniques remedy this issue by selecting and prioritizing a subset of test cases that are (1) sufficient to test new changes while accounting for their side effects, and (2) able to detect faults as early as possible. These techniques often rely on a mixture of code coverage analysis (e.g., [3]), heuristics based on test execution history (e.g., [4]), and domain-specific heuristics and rules (e.g., [3]). Further, some researchers (e.g., [5], [6]) have relied on machine learning (ML) techniques in order to learn, by combining all information sources, optimal selection and prioritization heuristics. This work provides a good basis on which to address the challenges of CI regression testing. However, existing approaches must still be improved to deal with the dynamic nature and timing constraints of CI. In general, applicable test case selection and prioritization techniques must be significantly faster than the run-them-all approach to be beneficial. While the same condition holds in the context of CI, such techniques should furthermore be fast enough to avoid delays in the typically quick build cycles, as this is the main justification for CI.

Any ML-based solution for test case prioritization in the context of CI needs to handle large amounts of historical data (e.g., test case and code change history) and adapt continuously to changes in the system and test suites, reflected in newly collected data. While supervised ML techniques can deal with abundant data, their continuous adaptation to new data is impractical and time-consuming. More specifically, the majority of current ML techniques are restricted to the classical batch setting that assumes the full data set is available prior to training, do not allow incremental learning (i.e., continuous integration of new data into already constructed models) but instead regularly reconstruct new

- *Mojtaba Bagherzadeh is with the School of EECS, University of Ottawa, Ottawa, ON K1N 6N5, Canada. E-mail: m.bagherzadeh@uottawa.ca.*
- *Nafiseh Kahani is with the Department of Systems and Computer Engineering, Carleton University, Ottawa, ON K1S 5B6, Canada. She contributed to this work mainly during her postdoctoral fellowship at the School of EECS, University of Ottawa, Ottawa, ON K1N 6N5, Canada. E-mail: kahani@sce.carleton.ca.*
- *Lionel Briand is with the School of EECS, University of Ottawa, Ottawa, ON K1N 6N5, Canada, and also with the SnT Centre for Security, Reliability and Trust, University of Luxembourg, 4365 Esch-sur-Alzette, Luxembourg. E-mail: lionel.briand@uni.lu.*

models from scratch. This is not only very time-consuming but also leads to potentially outdated models. For example, MART [6], the ML technique reported to be the most accurate for test case prioritization, does not support incremental learning because it is an ensemble model of boosted regression trees that is designed for static data [7].

Recently, Reinforcement Learning (RL) has shown great potential in various challenging scenarios that require continuous adaptation, such as game playing [8], real-time ads bidding [9], and recommender systems [10]. Inspired by this line of work and some initial and partial efforts in supporting test case prioritization with RL techniques, we perform here a *comprehensive* investigation of RL-based test case prioritization. To this end, taking test case prioritization as a ranking problem, we model the sequential interactions between the CI environment and a test case prioritization agent as an RL problem, guided by three different ranking models from information retrieval [11]: pairwise, listwise, and pointwise ranking. We then rely on carefully selected and tailored state-of-the-art RL techniques to automatically and continuously learn a test case prioritization strategy, whose objective is to be as close as possible to the optimal one. In particular, we introduce a CI environment simulator (i.e., replayer of test execution history) based on the three ranking models, which can be used to train model parameters using available and continuously incoming test execution history from previous CI cycles, in order to prioritize test cases in subsequent cycles. The training process is adaptive in the sense that the agent is provided feedback at the end of each cycle, by replaying the execution logs of test cases, to ensure that the agent policy is efficiently and continuously adapting to changes in the system and regression test suite. Existing results regarding RL, however, show that, in terms of accuracy, it does not fare nearly as well as the best supervised ML algorithms, e.g., MART [6]. Our main objective is therefore to benefit from the practical advantages of RL while at least retaining the prioritization accuracy of the best ML techniques.

We have conducted extensive experiments using a variety of carefully selected RL configurations (ranking model and RL algorithm) based on eight publicly available datasets, two of them containing only execution history while the remaining six are augmented with light-weight code features. We refer to the former and latter datasets as "simple" and "enriched", respectively. The results show that, for enriched datasets, the best configurations bring a significant ranking accuracy improvement compared not only with previous RL-based work, but also with MART. Further, though the accuracy is inadequate for simple datasets regardless of the employed RL technique, we reach high accuracy for all enriched datasets, leading to test case prioritization policies that are close to the pre-determined optimal policies for each dataset. Differences in training time across configurations, though significant, are not practically relevant in our context. Such results suggest that applying RL in practice would be beneficial when relying on adequate datasets, going beyond test execution history.

To summarize, our work makes the following contributions towards effective and scalable test case prioritization in the context of CI.

- A *comprehensive* set of solutions for the modeling of test case prioritization as an RL problem, including

algorithms that precisely describe how RL can be used for each of the three ranking models from information retrieval [11], in the context of test case prioritization and CI. These algorithms are then implemented using carefully selected, state-of-the-art RL techniques. This builds on previous work that takes a partial approach regarding the modeling, training, and implementation of RL: (1) it only uses the pointwise ranking model, (2) it relies on a small subset of RL techniques that seem to deviate from the standard, state-of-the-art algorithms provided by modern libraries [12], [13], [14]. Our work is the first that recasts pairwise and listwise ranking as an RL problem for test case prioritization. Our evaluation reveals that combining previously unused, state-of-the-art RL algorithms with pairwise ranking, results in the most accurate approach for test case prioritization.

- Extensive experiments with a comprehensive set of carefully selected, state-of-the-art RL algorithms based on the proposed ranking models. Existing work, which we compare against, only evaluates a small subset of non-standard RL implementations based on a pointwise ranking model; in contrast, our approach evaluates 21 different RL configurations. Further, as described in Section 5, past empirical studies on this topic have a number of issues that we attempt to address to provide more realistic results.

A comprehensive set of solutions for the modeling of test case prioritization as an RL problem, including algorithms based on three ranking models. These algorithms are then implemented using state-of-the-art RL techniques.

The rest of this paper is organized as follows. In Section 2, we define the test case prioritization problem, describe a running example, and provide background information on RL. We review related work in Section 3 and presents three approaches for modeling test case prioritization as an RL problem in Section 4. We present our evaluation approach and results in Section 5 and conclude the paper in Section 6.

2 BACKGROUND

In this section, we describe the terms and notations we use to define the test prioritization problem in our work, and describe the RL models we rely on to support test case prioritization.

2.1 Test Case Prioritization

Regression testing of a new software release is an essential software quality assurance activity. However, the regression testing of a software system with a large code base often requires the execution of a large number of test cases, which is time-consuming and resource-intensive. Test case prioritization aims to find an optimal ordering for the test case executions to detect faults as early as possible. Thus, executing a small fraction of a prioritized test suite may reduce the cost and time of testing while detecting most of the faults. In this work, our focus is regression test cases prioritization in the context of Continuous Integration (CI) of a software system where at each CI cycle, the system is built

TABLE 1
Code-Based Features, Adopted From [6]

Type	Features	Description
Program size	AvgLine, AvgLineBlank, AvgLineCode, AvgLineComment, CountDeclFunction, CountLine, CountLineBlank, CountLineCode, CountLineCodeDecl, CountLineCodeExe, CountLineComment, CountSemicolon, CountStmt, CountStmtDecl, CountStmtExe, RatioCommentToCode	Features related to the amount of lines of code, declarations, statements, and files
McCabe's cyclomatic complexity	AvgCyclomatic, AvgCyclomaticModified, AvgCyclomaticStrict, AvgEssential, MaxCyclomatic, MaxCyclomaticModified, MaxCyclomaticStrict, MaxEssential, MaxNesting, SumCyclomatic, SumCyclomaticModified, SumCyclomaticStrict, SumEssential	Features related to the control flow graph of functions and methods
Object oriented metrics	CountDeclClass, CountDeclClassMethod, CountDeclClassVariable, CountDeclExecutableUnit, CountDeclInstanceMethod, CountDeclInstanceVariable, CountDeclMethod, CountDeclMethodDefault, CountDeclMethodPrivate, CountDeclMethodProtected, CountDeclMethodPublic	Features based on object-oriented constructs

and released upon successful testing, including regression testing.

Definition 1 (CI and CI Cycles). *We capture the CI history of a software system as sequence of cycles $c_i, 1 < i < n$, where c_1 and c_n refer to the first and current cycles, respectively. A cycle c is a tuple $\langle T, f \rangle$, where T is a set of test cases, and f is a logical value that indicates whether or not the cycle has failed. The number of test cases in different cycles varies. A cycle can fail due to several reasons, including compilation errors or a test case failure. However, in this work, we are only interested in the latter and failed cycles are, in our experimental datasets, cycles with at least one failed test case.*

Definition 2 (Test Case Feature Records). *Each test case has two feature records: execution history and code base features. Execution history of a test case at cycle c_i is defined as a tuple $<v, e, h, a>$, where v shows the execution verdict of the test case at cycle c_i, e represents the execution time of the test case at cycle c_i, h is a sequence of verdicts that shows the test case verdicts at prior cycles, i.e., $c_j, 1 < j \le i - 1$, and a represents the test age capturing when the test case was first introduced. The execution history for each test case contains a record of executions over previous cycles. The execution verdict is considered either 1 if the test case has failed, or 0 if it has passed. Similar to previous work [15], we assume the execution time (e) of the test case to be the average of its previous execution times.*

The accuracy of all ML techniques largely depends on the features they use. Solely relying on execution history, thus ignoring many relevant code-based features, such as the complexity of changes or test cases, can lead to low prioritization accuracy. Thus, we adopt code-based features from the literature [6] that are extracted using light-weight and incremental static analysis and repository mining techniques, and are thus applicable in a CI context. Code-based features such as Line of Code (LoC) are relevant predictors of test case execution time and failure occurrences. For instance, if tests t1 and t2 target source files f1 and f2 respectively, and f1 has more LoC, then the execution of t1 is likely to take longer than that of t2, because t1 targets a more complex source file. We can make a similar argument about the probability of failure.

Table 1 lists the code-based features calculated for each test case, based on the source code classes that are exercised (covered) by test execution. Assuming dependency and coverage data for source code classes is available, Bertolino et al. [6] use the following four-step process to relate code-based features to test cases and create a vector for each test case in each CI build.

(1) *The changed classes in the build are identified.*
(2) *All impacted files are extracted from the dependency database based on the changed classes (output of step 1).*
(3) *For each test case, a subset of impacted classes (output of step 2) covered by its execution are identified.*
(4) *For each test case, code-based features are calculated based on covered classes (output of step 3). When a test case covers more than one file, the features are calculated based on all of the covered classes by adding their values.*

Dependency and coverage data is collected from the source code of both test cases and the system under test using static analysis techniques (i.e., more specifically using Understand [16]). Then such data is updated for each build based on new changes. Impact analysis and incremental updates enable efficient coverage and dependency analysis (as discussed in Section 5.1). Though static analysis tends to overestimate coverage and dependencies, such information can help improve the accuracy of ML models.

Definition 3 (Test Case Prioritization). *Regardless of underlying techniques for test case prioritization, we assume this to be a ranking function that takes in input a set of test cases' features and returns an ordered sequence in which the position (index) of the test cases shows their priority for execution, i.e., the test case with the lowest index (rank) gets executed first.*

Definition 4 (Optimal Ranking). *Given a set of n test cases (T), the ranking function can produce $n!$ distinct ordered sequences. We define the optimal order (s_o) of a set of test cases T as a sequence in which:*
$$\forall t_1, t_2 \in T,$$
$$idx(s_o, t_1) < idx(s_o, t_2) \Leftrightarrow t_1.v > t_2.v \text{ or}$$
$$(t_1.v = t_2.v \text{ and } t_1.e <= t_2.e),$$
where function $idx(s, t)$ returns the index (rank) of a test case t in sequence s.

The above condition implies that, in the optimal ranking of the test cases, (1) all failing test cases (their verdict is one) are

Fig. 1. An artificial example of CI.

Fig. 2. Optimal rankings for the illustrative example (Fig. 1).

executed before passing test cases, (2) test cases with lower execution time are executed earlier than other test cases with the same verdict.

We use the optimal order s_o as a reference ranking (ground truth) and our goal is to find a test case ranking function whose output is as close as possible to s_o. Note that we give higher importance to the verdict than the execution time. However, depending on the context, the optimal ranking can be tuned. For instance, if one knows with confidence that failures are very rare, the optimal ranking can be tuned to give higher importance to execution time rather than the verdict.

2.2 Illustrative Example

Fig. 1 shows an artificial CI example in which only history features of test cases are included in the interest of space. The example shows three completed CI cycles, each of which contains a few test cases. As shown, (1) the number of test cases varies across cycles, (2) the execution history of a test case at a specific cycle contains previous execution verdicts, and (3) the age of test cases is incremented after each cycle completion.

An optimal ranking for a specific cycle is estimated based on test execution history. This is because test execution times and verdicts are unknown until all test case executions are completed and; therefore, the optimal ranking is unknown.

Based on our illustrative example above, we provide optimal rankings for all cycles in Fig. 2.

2.3 Reinforcement Learning (RL)

In RL, an agent interacts with its environment through the use of observations (states), actions, and rewards. At each interaction step t, the agent receives some representation of the environment's state as input, $S_t \in S$, where S is the set of possible states. Based on the perceived state, the agent chooses an action, $A_t \in A(S_t)$, where $A(S_t)$ is the set of actions available in state S_t, to generate as output. The action selection is based on either a learned or an exploration policy. As a result, the agent receives feedback in terms of reward, which rates the performance of its previous action.

State-of-the-art RL techniques can be classified based on the following properties:

Model-Based versus Model-Free. In model-free RL algorithms, it is assumed that an agent neither has any prior knowledge of the environment (a black-box environment) nor attempts to learn the environment dynamics. In other words, the agent does not know beforehand how the environment reacts to possible actions, or what the next state and reward will be before taking an action. So, the agent needs to interact with the environment and observe its responses to devise an optimal policy for selecting an action.
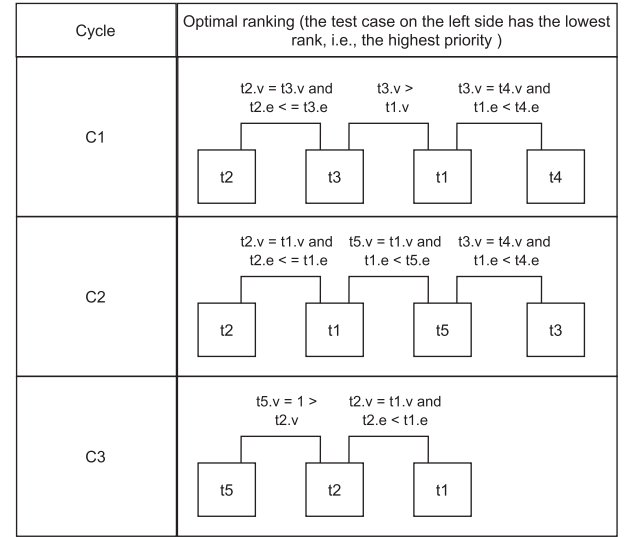
As mentioned earlier, the execution time and results of test case executions at a given CI cycle are unknown before their execution. Therefore, we only use model-free RL algorithms for test case prioritization.

Value Based, Policy Based, and Actor-Critic Learning. Assuming that the Q-value is a measure of the expected reward in a state for a given action, value-based methods estimate the Q-value of possible actions for a given state and select the action with the highest value. An example of value-based methods is the Q-learning algorithm [17] that, in its simplest form, uses a Q-table and the Bellman equation to estimate the Q-value. In its more advanced form (DQN), it uses a deep neural network to estimate the Q-value [18]. Policy-based methods directly search for an optimal policy. Typically, a parameterized initial policy is chosen, whose parameters are updated to maximize the expected return using either gradient-based or gradient-free optimization. An example of policy-based methods is the REINFORCE algorithm [19].

Each of the above methods has drawbacks and benefits. More importantly, value-based methods are often sampling efficient, but the convergence is guaranteed in very limited settings that often requires extensive hyperparameter tuning. On the contrary, policy-based methods are stable but sample inefficient, i.e, convergence is guaranteed but at a very slow rate [20], [21], [22]. Actor-critic methods aim at combining the strong points of actor-only (value-based) and critic-only (policy-based) methods. The critic uses an approximation architecture and simulation to learn a value function, which is then used to update the actor's policy parameters. Such methods have desirable and faster convergence properties compared to value-based and policy-search based methods [21]. In this work, we do not exclude any algorithm based on their learning method because there is no evidence regarding the superiority of a certain method in all contexts.

Action and Observation Space. The action space specifies how the agent can act on its environment, while the observation space specifies what the agent can know about its environment. The latter is referred to as feature space in ML. Both the observation and action spaces come in discrete

and continuous forms. In the simplest form, an observation can only be a real number (e.g., the position of the agent) but it can also be more complex and high-dimensional (e.g., RGB matrix of observed pixel values). With a discrete action space, the agent decides which distinct action to perform from a finite action set, whereas with a continuous action space, actions are predicted and expressed as a real-valued vector. While most of the RL algorithms do not impose constraints on the observation space, not all of them support both discrete and continuous action spaces, and therefore their application is restricted according to the problem's action space. Further, it is possible for the action space to be a vector of continuous or discrete values. As we will discuss later (Section 4), we use three different approaches for modeling test case prioritization, each of them having a different form of action space that limits our choice of algorithms.

On-Policy versus Off-Policy. There are two types of policy learning methods, namely on-policy (e.g., SARSA) and off-policy. In on-policy learning, the agent attempts to learn a policy that is close to the exploration strategy, i.e, the learned policy is influenced by the exploration strategy. While, in off-policy learning (e.g., Q-learning), the learned policy is independent of the exploration strategy, i.e., exploration during the learning phase is not based on the learned policy [23].

2.4 State-of-the-Art RL Algorithms and Frameworks

Several model-free RL algorithms have been proposed over the last few years that advance the state of the art, e.g., Deep Deterministic Policy Gradient (DDPG) [25], Deep Q-Networks (DQN), Advantage Actor-Critic (A2C) [26]. Further, several open-source research frameworks provide reusable implementations of the state-of-the-art algorithms, e.g., Acme [14], Stable Baselines [12], and OpenAI Baselines [13]. In this work, our focus is on the application of RL techniques rather than devising new RL techniques. Thus, we rely on the *Stable Baselines framework* and state-of-the-art algorithms which it provides. *Stable Baselines* is the improved version of *OpenAI Baselines*, with more comprehensive documentation and support for more algorithms compared to other frameworks. A list of the supported algorithms that match our problem as we discuss in Sections 4 and 5, as well as their properties, are shown in Table 2. Note that all of the above algorithms use deep neural networks (DNNs) to capture policies.

3 RELATED WORK

Test prioritization for regression testing has long been an active area of research [33], [34], [35]. Existing work can be categorized into two groups: heuristic-based and ML-based test prioritization.

Heuristic-Based Test Prioritization. The proposed methods of this group have typically used heuristics based on information such as code coverage [3], [36], models [37], [38], history [4], [39], [40], and requirements [41], [42]. The main drawback of these methods, especially in a CI context, is that they are not adaptive to quickly changing environments.

TABLE 2
Model-Free, State-of-the-Art RL Algorithms

Algo.	Lear.	On/Off	Act.
DQN [24]	Value	Off-policy	Dis
DDPG [25]	Policy	Off-policy	Cont
A2C [26]	Actor-Critic	On-policy	Both
ACER [27]	Actor-Critic	Off-policy	Dis
ACKTR [28]	Actor-Critic	On-policy	Both
TD3 [29]	Policy	Off-policy	Cont
SAC [30]	Actor-Critic	Off-policy	Cont
PPO1 [31]	Actor-Critic	On-policy	Both
PPO2 [31]	Actor-Critic	On-policy	Both
TRPO [32]	Actor-Critic	On-policy	Both

Cont: *continuous,* Dis: *discrete,* Both: *discrete and continuous.*

A large body of existing work focuses on using code coverage information and the analysis of code modifications to order test cases. Coverage-based techniques stem from the idea that early maximization of structural coverage can increase the chances of early maximization of fault detection [4]. Some of the structural coverage measures include statement coverage [36], functions/methods coverage [3], and modified condition/decision coverage [43]. Rothermel *et al.* [36] presented several approaches for prioritizing test cases and reported empirical results measuring the effectiveness of these approaches. Overall, the coverage-based work can be grouped into two groups: total requirement coverage and additional requirement coverage [44]. The former orders test cases in decreasing order of the number of statements they cover. The latter prioritize test cases in decreasing order of the number of additional statements they cover, that is statements that have not yet been covered by the previously executed test cases.

Coverage information can be collected either by static or dynamic analysis. Lightweight static analysis techniques overestimate the coverage data and are not accurate [45]. More thorough static analysis techniques (e.g., static analysis with reflection support) can significantly improve the accuracy of coverage information, but their high computation cost renders them impractical [45]. Similarly, dynamic analysis techniques are difficult or even impossible to apply in practice, and more specifically so in a CI context. The reasons are discussed in several papers [15], [46], [47], [48], [49] and summarized below:

- Computation Overhead: Code analysis and instrumentation take a long time to execute for a large codebase [46], [48], [49]. As reported in [49], running a code instrumentation tool at each milestone on the codebase of Google and collecting code coverage data would impose too large an overhead to be practical.
- Applicability: They are applicable only to complete sets of test cases, as they search in the space of all test cases and select/prioritize them to reach either maximum coverage or defect detection, or minimum execution cost [46]. Also, the extraction of code coverage requires traceability between code and test cases, the information that is not always available or easily accessible with system tests (i.e., black-box testing) [47]. Further, the non-ML based techniques

are often language and platform-dependent, which leads to more customization and effort.

- Maintainability: Typically, high code change rates, in actively developed projects, quickly render code coverage data obsolete, requiring frequent updates [48], [49].

Several researchers [38] used executable system models to select and generate test cases related to the modified parts of the system. Models are an abstraction of the actual system. Such abstractions make the model execution for the whole test suite relatively inexpensive and fast compared to the execution of the actual system [50]. However, the source code may change over time, resulting in the need to update the models to reflect the changes. Such updates create overhead when relying on model-based approaches. Also, models are often extracted using source code analysis; therefore, they inherit the drawbacks of code-based approaches. Korel *et al.* [38] presented a model-based prioritization approach in which the original and modified system models, along with the information about the system model and its behavior, are used to prioritize test cases. While code-based coverage approaches are more precise compared to model-based ones, they introduce practical challenges in terms of complexity and computational overhead to collect and analyze code coverage information.

History-based approaches rank tests based on past test execution data. These approaches are based on the idea that past test case failures are a good predictor of test cases with a high probability of failure in new releases. Kim and Porter [4] proposed a history-based approach that calculates ranking scores based on the average of past execution results. Park *et al.* [40] proposed a history-based approach to analyze the impact of test costs and severity of detected defects in a test suite on test prioritization. Noor and Hemmati [51] defined a class of quality metrics that estimate test case quality using their similarity to the previously failing test cases from previous releases. Their results showed that adding similarity-based test quality metrics along with traditional test quality metrics can improve the test prioritization results. History-based approaches are less expensive than coverage-based and model-based approaches. However, learning optimal test case prioritization policies only based on test execution history seems difficult, specifically for complex software systems. Also, they may not be well adapted to continuously changing testing environments with frequent changes in code and test suites. In our work, we also rely on historical information, with the differences that (1) our RL-based solution is seamlessly adaptive and can therefore deal with the dynamic nature of CI (2) we use enriched execution history with code-based features to improve the accuracy of the prioritization.

Srikanth *et al.* [41] proposed a model for system-level test case prioritization from software requirement specifications. They mapped test cases to software requirements, and then prioritized the test cases based on four factors including requirements volatility, customer priority, implementation complexity, and fault-proneness of the requirements. Similar work [42] proposed a system-level technique for prioritization based on requirements according to four factors including customer assigned priority of requirements, developer-perceived implementation complexity, requirement volatility and fault proneness of the requirements.

Some work [46], [52] proposed heuristic-based test prioritization methods tailored to CI environments. Marijan *et al.* [52] proposed a weighted history-based test prioritization approach called ROCKET, which orders test cases based on historical failure data, test execution time, and domain-specific heuristics. Elbaum *et al.* [46] presented a test selection approach at the pre-submit stage that uses time windows to track how recently test cases have been executed and revealed failures. To increase the cost-effectiveness of testing, they performed test prioritization based on the specified windows to prioritize test cases that must be executed during subsequent post-submit testing.

In contrast with these techniques, our approach uses RL to prioritize test cases. Relying on RL makes our approach seamlessly adaptive to the changing CI environment. Also, by combining various data sources (e.g., coverage, failures, execution time), we may be able to build more accurate prioritization models.

ML-Based Test Prioritization Techniques. Work [53] in this category investigates the application of ML techniques to test prioritization. The motivation is to integrate data from different sources of information into accurate prediction models. Results have shown that ML techniques can provide noticeably promising results in test selection and prioritization [53].

Several approaches studied the effectiveness of clustering for test prioritization. Carlson *et al.* [54] cluster test cases based on code coverage, code complexity, and fault history data. Lenz *et al.* [55] grouped test cases into functional clusters derived by executing some example test cases. The test results and clusters feed ML classifiers, which produce sets of rules to classify the test cases. The rules were used to support various tasks, including test case prioritization.

Past research [5], [6], [56], [57] has also proposed a number of supervised ML techniques that reduced test prioritization to a ranking problem. Tonella *et al.* [56] proposed a pairwise ranking algorithm to rank test cases based on coverage, complexity metrics and historical data. Busjaeger and Xie [5] introduced a listwise method based on ML combined with multiple existing heuristic techniques to prioritize test cases in industrial CI environments. They used features include coverage data, test file path similarity and test content similarity, failure history, and test age. Lachman *et al.* [57] applied SVM Rank to black-box prioritization starting from test cases and failure reports in natural language (NL).

However, supervised and unsupervised ML techniques tend to be impractical in a CI context when prediction models need to continuously and quickly adapt to new data, reflecting changes in the system and test suites. To deal with this issue, recent work has investigated the application of RL. In an initial attempt to apply one RL algorithm to test case prioritization in CI environments [15], Spieker *et al.* prioritize test cases according to their execution time and previous execution and failure history. Their work is based on the pointwise ranking model and only uses the Q-learning RL algorithm. In contrast, in this work, we perform a comprehensive investigation of RL techniques by guiding the RL agent according to three different ranking models: pairwise, listwise, and pointwise ranking.

In very recent work, Bertolino *et al.* [6] analyze the performance of ten ML algorithms, including three RL algorithms, for test prioritization in CI. Through an experimental analysis, they show that Non-RL-based approaches to test case prioritization are more affected by code changes, while the RL-based algorithms are more robust. Similar to Spieker *et al.* [15], Bertolino's application of RL is based on the pointwise ranking model. Their results show that their specific RL configuration is significantly less accurate compared to the best ranking algorithms based on supervised learning (e.g., MART). Further, the above RL-based work only experiment with a small subset of RL implementations that differ from state-of-the-art algorithms provided by modern libraries [12], [13].

This paper builds on past work by applying RL to CI regression testing. It does so by investigating all ranking models: pointwise, pairwise, and listwise. Further, for each model, we experiment with all available and applicable state-of-the-art RL algorithms, as it is difficult to a priori determine which ones will work better in a CI regression testing context. Thus, we increase our chances of obtaining accuracy results that are close to or better than the best supervised learning techniques, e.g., MART, while getting the practical benefits of RL.

4 REINFORCEMENT LEARNING FOR TEST CASE PRIORITIZATION

An Overview. We aim to develop an RL-based solution for the prioritization of test cases in the context of CI. Most existing test case prioritization solutions consider the prioritization procedure as a static process and prioritize test cases following a fixed strategy that is defined based on either heuristics or supervised ML techniques [33]. Here, we investigate variants of a prioritization approach capable of continuously adapting and improving its strategy as a result of its interactions with the CI environment. We model the sequential interactions between CI and test case prioritization as an RL problem and rely on state-of-the-art RL techniques to automatically and continuously learn a test case prioritization strategy that is as close as possible to the optimal one, assuming a pre-determined optimal ranking as the ground truth. In particular, we introduce a CI environment simulator, which can be used to train the agent offline using the available test execution history before applying and updating the model online. In other words, we train an RL agent based on test execution history and code-based features from previous cycles in order to prioritize test cases in subsequent cycles. The training process is adaptive in the sense that the agent can be provided with feedback at the end of each cycle or, when the agent accuracy is below a certain threshold, execution logs of test cases can be replayed to ensure the agent policy is efficiently and continuously adapting to changes in the system and regression test suite.

We rely on a typical approach for developing an RL solution in a specific context: (1) devise algorithms that precisely describe how RL can be used for each ranking model, in the context of test case prioritization and CI via replaying the test cases' execution history, (2) train an RL agent using carefully-selected, state-of-the-art RL techniques, as discussed in Section 2. One important goal in this paper is to

be as comprehensive as possible in investigating alternatives. Next, we discuss possible solutions for creating an RL environment for test case prioritization, with a focus on the formalization of action and observation spaces, reward functions, and interactions between the environment and the RL agent. We then describe the way RL techniques can be applied to train an agent in our context and discuss how the RL agent can be integrated into CI environments.

4.1 Creation of the RL Environment

As discussed in Section 2.3, the RL agent and environment interact by passing the observation, reward, and action. The typical flow of the interaction is shown in Algorithm 1. First, the agent is given an initial observation by the environment. Then an episode starts, during which the agent perceives the current observation and selects an action based on the exploration strategy, which varies according to the underlying algorithm, e.g., Q-learning uses the epsilon-greedy ($\epsilon-$greedy) exploration method. An episode is a sequence of states and actions that takes an RL agent from an initial state to a final state, in which the agent task is completed.

The selected action is passed to the environment that applies the action and returns a new observation and reward. The agent takes the reward and observation into account and updates its policy according to the underlying RL technique and most particularly the learning method (policy, value, or actor-critic based), as discussed in Section 2. The episode ends when the task is done, regardless of success or failure. The end condition depends on the nature of the task, e.g., for an agent that plays a game, an episode ends when the game ends.

Algorithm 1. A Training Episode of an RL Agent

---
1 Let done be False \\ a flag to capture the end of episode
2 Let obs be a valid initial observation
3
4 **while not** done \\ an episode
5   action=predict(obs) \\ the agent select an action
6   done, reward, obs = step(action) \\ applying the action
7   updateAgentPolicy() \\ agent reinforces its policy

To map the test case prioritization problem to RL, the details of the above-mentioned interactions (i.e., observation, action, reward, and end condition of an episode) need to be defined properly. Assuming test case prioritization to be a ranking function (Definition 3), the interaction details can be defined based on ranking models from the information retrieval field [11]: pointwise, pairwise, and listwise. A pointwise ranking approach takes the features of a single document and uses a prediction model to provide a relevance score for this document. The final ranking is achieved by simply sorting the documents according to these predicted scores. For pointwise approaches, the score for each document is independent from that of the other documents.

A pairwise approach orders a pair of documents at a time. Then, it uses all the ordered pairs to determine an

optimal order for all documents. Some of the most popular Learning-to-Rank algorithms are pairwise approaches [58], e.g., RankNet, LambdaRank and LambdaMART. Listwise ranking approaches consider a complete list of documents at once and assign a rank to each document relative to other documents.

Each ranking model has advantages and drawbacks in the context of test case prioritization, that will be further discussed below. Also, the interaction details between the *Agent* and *Environment* will differ based on the underlying ranking model. In the following, we discuss how each of the ranking models can be applied in the context of test case prioritization.

4.1.1 Listwise Solution

Algorithm 2 shows the details of an episode of listwise ranking that starts by setting the initial observation containing a vector of all test cases' features. As discussed in Section 2, the number of test cases varies in different cycles. However, RL, as other ML techniques, does not handle inputs of variable size. Therefore, the size of the observation space for a given CI system needs to be defined, based on the maximum number of test cases in a cycle, to allow for the trained agent to handle all cycles by using padding. For example, based on the illustrative example, the maximum number of test cases in a cycle is 4, while the number of test cases for cycle C_3 is 3. Thus, we add a dummy test case whose

---

Algorithm 2. RL-Based Listwise Ranking

```
 1 Input
 2 Let T be a @set@ of test cases
 3 Let s_o be the optimal ranking of T
 4
 5 Output
 6 s_e an order (a sequence) of test cases in T
 7
 8 Let rank be 0, done be False, @and@ reward be 0
 9 Let obs be a vectorized T # observation space
10 Let action be an integer with @range@ [0, |obs|-1]
11 episode() # ref. Algorithm 1
12
13 Function step(action)
14     reward = calc_reward(action)
15     if rank < |T|-1 and obs[action] is not dummy:
16        append obs[action] to s_e
17        obs[action] = a dummy test case # mark the test
           case as selected
18        rank = rank +1
19     else if rank = |T|-1
20        done = True
21     return done, reward, obs
22
23 Function calc_reward(action)
24     if obs[action] is dummy: # previously selected or
           padded
25        reward = 0
26     else
27        optimal_rank = idx(s_o, obs[action])
28        reward = 1- (norm(optimal_rank) - norm(action))^2
```

features are set to -1 into the feature record of C_3. We refer to the process of creating these dummy test cases as padding.

After preparing the initial observation, an episode is started, during which the agent selects the index of the test case with the highest priority (lowest rank) as an action. As shown in function *step* of Algorithm 2, the environment applies the action by (1) appending the selected test case to the output sequence (s_e), (2) updating the observation by replacing the feature record of the selected test case with the dummy test case to keep track of selected test cases, and (3) calculating a reward that is shown in function *calc_reward*.

Observation and Action Space. In the listwise ranking, the observation space grows linearly with the number of test cases and this increases the training and prediction time. In general, dealing with large observation and action spaces is one of the main challenges with RL [59], [60]. In our context, in which the feature record of a test case has at least four numeric fields (Definition 2), for a system with 1,000 test cases the observation space grows to 4,000 numeric fields (features). This is clearly a high-dimensional observation space, with each feature having a large range of possible numeric values. Thus, an enormous amount of training data is required to ensure that there are several samples for each combination of values to train the model. Coping with this kind of growth in feature space dimensionality is an open and active research area in machine learning [61].

Action Space. The action space in listwise ranking is a discrete value whose range is defined by the number of test cases. That value, the action, captures the test case with the highest priority. Similar to the observation space, the action space grows linearly with the number of test cases. This can also lead to scalability issues since the agent needs to evaluate all possible actions at each step. Thus, the larger the action space, the larger the training time. In general, existing RL techniques cannot handle large discrete spaces [59], [60].

As an alternative, it would be possible to define the action space of the listwise ranking as a vector of either discrete or continuous values, each of which representing the rank of a test case, and then train an agent to assign ranks of all test cases in one step. A vector of discrete or continuous values as an action space is supported by the existing RL algorithms (e.g., TRPO) and can be applied for a cycle with a small number of test cases. However, when the action space is large, training the agent (finding an optimal policy) is difficult. Some algorithms do not converge to the optimal policy, converge very slowly, or, in some cases, have prohibitive computational requirements [62]. We have performed an initial experiment using a vector of discrete values as action space and tried to train an agent on a cycle with 600 test cases that took more than six hours. As a result, in our experiments, we will adopt the first solution presented above, where the action space is a single discrete value.

Reward Function. To calculate reward, we take the optimal test case ranking (Definition 4) as the reference and compare the assigned rank of each test case with respect to its rank in the optimal ranking. As shown in function *calc_reward* of Algorithm 2, the reward function calculates the reward, as a value within [0,1], for the selected action (index

of a test case) based on its deviation from the optimal ranking. The agent gets the highest possible reward when the rank assigned to a test case is equal to the optimal one. The smaller the distance between the RL and optimal rankings, the higher the reward. Further, the agent is given the lowest reward (zero) when the agent selects a dummy test case, resulting in either from the test case having already been selected or padded.

Overall, the listwise ranking is easy to model. However, the high dimensionality of the action and observation spaces cause scalability issues. We provide more details on the performance of this ranking model in Section 5.

4.1.2 Pointwise Solution

Algorithm 3 shows the details of an episode of pointwise ranking that starts by converting the set of test cases to a sequence and then setting the initial observation to the first test case of the sequence. It then begins a training episode, during which the agent determines a score for the test case that is a real number between 0 and 1. As shown in function *step* of Algorithm 3, the environment applies the action by (1) saving the selected score in a temporary vector (tmp), (2) updating the observation by setting the next test case of the vector as observation, (3) calculating a reward that is shown in function *calc_reward*, and (4) at the end of episodes, sort all the test cases based on their assigned scores saved in tmp.

Algorithm 3. RL-Based Pointwise Ranking

```
 1 Input
 2 Let T be a @set@ of test cases
 3 Let sₒ be the optimal ranking of T
 4
 5 Output
 6 sₑ an order (a sequence) of test cases in T
 7
 8 Let sₑ be a random sequence of test cases in T
 9 Let index be 0, done be False, @and@ reward be 0
10 Let obs be a vector of [sₑ[index]]
11 Let tmp be an empty vector # keeps ranks
12 Let action be a real number with a @range@ (0,1]
13 episode () ## ref. Algorithm 1
14
15 Function step(action)
16     reward=calc_reward(action)
17     add action into tmp
18     if index < |sₑ|−1
19       index = index+1
20     else
21       done = True
22       sort sₑ based on their ranks in tmp
23     obs = sₑ[index]
24     return done, reward, obs
25
26 Function calc_reward(action)
27     Let optimal_rank be the position of sₑ[index] in
         optimal ranking
28     reward = 1− (norm(optimal_rank) − action)²
```

Observation and Action Space. An observation here is the feature record of a single test case, which is much smaller than an observation for the listwise approach. The action space is a continuous range between 0 and 1, that is the test case's score, a real number based on which test cases are ranked.

Reward Function. Similar to listwise, we take the optimal ranking as the reference and calculate the reward based on the distance between the assigned and optimal ranks. However, since each test case is scored individually during training, the final rank of the test cases is not known until the end of the training episode. Thus, as shown in function *calc_reward* of Algorithm 3, we compute the distance by normalizing the optimal rank of test cases. Since the normalized optimal and assigned score values both range between 0 and 1, their difference provides meaningful feedback to the agent. An alternative is to only provide the reward at the end of the episode. However, this leads to the sparse reward issue [63] and makes the training of the agent inefficient.

Overall, the pointwise ranking is easier to model than listwise and the dimensionality of its observation space is much smaller compared to listwise. We provide more details on the performance of this ranking model in Section 5.

4.1.3 Pairwise Solution

Algorithm 4 shows the details of an episode of pairwise ranking that starts by setting the initial observation to a pair of test cases. It then starts a training episode, during which the agent selects either 0 or 1, the former denoting that the first test case in the pair has higher priority (lower rank). In general, ranking a pair is based on a comparison operator is the essential building block of the sorting algorithms, such as selection and merge sort [64]. Thus, based on the ranking of a pair, the environment can apply any of the sorting algorithms to prioritize the test cases. For example, as shown in function *step* in Algorithm 4, the environment applies the selection sort algorithm [64] to sort all the test cases based on ranking pairs. Thus, it updates the observation based on a mechanism that the selection sort prescribes. That is, the list is divided into two parts (idx_0 is the splitter), the sorted part at the left of idx_0 (indices below idx_0) and the unsorted part at the right of idx_0. At each iteration of the sort, it finds the test cases with the highest priority and changes its position to idx_0 and then increases idx_0 and repeats this process until all test cases are sorted. The environment also calculates the rewards as shown by *calc_reward* in Algorithm 4.

The required steps of each training episode are determined by the complexity order of the applied sorting algorithms. For example, the complexity order of the selection sort is quadratic in the worst case and requires n^2 comparisons where n is the number of test cases. Therefore, each episode of the agent training requires n^2 steps to prioritize n test cases. While this is manageable for cycles with a small number of test cases, it causes very long-running episodes for cycles with a large number of test cases, e.g., for 10,000 test cases, an episode requires around 50 million steps to be completed. To alleviate this problem, we can adopt sorting algorithms with a lower complexity such as merge sort [64], the complexity of which is linearithmic ($n \, log(n)$). Thus, we also defined another version of the pairwise environment based on the merge sort algorithm, which is similar to

Algorithm 4. Merge sort is actually used in our experiments but it is quite long to describe ($>$60 lines of code) and is not necessary to understand the fundamentals of our approach. The interested reader can refer to the source code.[1]

Algorithm 4. RL-Based Pairwise Ranking

```
 1 Input
 2 Let T be a @set@ of test cases
 3 Let sₒ be the optimal ranking of T
 4
 5 Output
 6 sₑ an order (a sequence) of test cases in T
 7
 8 Set sequence sₑ to a be random order of test cases
   in T
 9 Let idx₀ be 0, idx₁ be 1, done be False, @and@
   reward be 0
10 Let obs be a vector [sₑ[idx₀] , sₑ[idx₁]] # a pair
11 episode()
12
13 Function step(action)
14    reward = calc_reward(action)
15    # the following lines performs the selection
      sort in a stepped way
16    if action == 1:
17       swap(sₑ, idx₀, idx₁)
18    if idx₁ < |sₑ|-1:
19       idx₁ = idx₁ + 1
20    else if (idx₁ == |sₑ| - 1) and (idx₀ < |sₑ| - 2):
21       idx₀ = idx₀ + 1
22       idx₁ = idx₀ + 1
23    else
24       done = True
25    obs = vector of [sₑ[index₀] , sₑ[index₁]]
26    return done, reward, obs
27
28
29 Function calc_reward(action)
30    Let sel_test_case be obs[action] and nonsel_test_case be
      obs[1 − action]
31    if sel_test_case.v > nonsel_test_case.v:
32       reward = 1
33    else if sel_test_case.v < nonsel_test_case.v:
34       reward = 0
35    else if sel_test_case.e <= nonsel_test_case.e:
36       reward = 0.5
37    else
38       reward = 0
```

Observation and Action Space. An observation (state) is a pair of test case feature records and is therefore much smaller than for listwise, but twice as large as pointwise. The action space is simply the set 0,1, which is much simpler than the action spaces of both listwise and pointwise.

Reward Function. If the agent gives higher priority to the only failed test case in the pair, it receives the highest reward (1). Otherwise, it receives no reward (0). Also, if both test cases in a pair have the same verdicts, then the agent receives 0.5 as the reward when it gives higher

priority to the test case with less execution time. Otherwise, the agent receives no reward (0). Different reward values are used to distinguish the actions according to their level of desirability, for example, to signal to the agent that higher priority for failed tests is most desirable, followed by lower execution times for test cases with identical verdicts. We assign intermediary reward values, between 0 and 1, when ranking test cases with the same verdict, since the detection of failures is more important than saving computation resources. Nevertheless, due to the very small failure rate of the subjects (Section 5) used in our experiments, we use a relatively large reward value (0.5) to signal the agent about the importance of ranking test case with the same verdicts, a frequent task when failures are rare.

Overall, the action space and observation space of pairwise ranking are relatively small, which is expected to improve the scalability of RL. On the other hand, it is a priori unclear whether learning pairwise comparisons is sufficient to reach a good ranking accuracy. Indeed, to obtain a complete ranking we rely on sorting algorithms based on imperfect pairwise comparisons. We provide empirical results regarding the performance of this ranking model in Section 5.

4.2 Train an RL Agent

We use state-of-the-art RL algorithms, listed in Table 2, to train an agent. However, their applicability is limited by the type of their action space. More specifically, *A2C, PPO1, PPO2,* and *TRPO* are applied to all of the ranking models, *DQN, ACKTR,* and *ACER* are applied only to pairwise and listwise ranking, and *DDPG, TD3,* and *SAC* are applied only to pointwise ranking.

In the context of many systems, especially safety-critical systems, an RL agent cannot be trained online directly via interacting with a real environment. Instead, the agent is trained using a simulator that models the environment or replays the logs of system executions [59]. In the latter case, after training an initial agent, the agent is deployed into the real environment. However, it is evaluated and trained offline based on new incoming logs in an iterative manner. Such offline training fits the CI context in which the execution logs of test cases are available and can be replayed to train an RL agent at the end of each CI cycle.

Regarding offline training, at the beginning, we train the agent based on available execution history and then, for each cycle, (1) we use the agent to rank test cases in the prediction mode (no training policy update is done while predicting), (2) we apply the ranking and capture the execution logs, and (3) we use the new logs to train the agent in an offline mode that allows the agent to adapt to new changes. The last step is fast since only the logs of one cycle are used for training, thus enabling offline training after each cycle.

4.3 Integration of an RL Agent into CI Environments

Fig. 3 shows how an RL agent can be integrated with the CI environment to prioritize regression test cases during a CI build. First, the agent needs to be trained based on the available data (test case execution history and related features extracted from the source code history) to reach a satisfactory accuracy. The trained agent is then deployed in the

---

1. https://github.com/moji1/tp_rl/blob/master/testCase_prioritization/PairWiseEnv.py
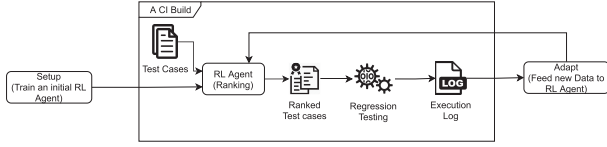
Fig. 3. Integrating an RL agent in a CI environment for test case prioritization.

production environment and can be invoked by passing test case features in each CI cycle to rank test cases. Test cases are executed according to their ranks during regression testing, and their execution logs are captured. At pre-determined times or when the RL agent's accuracy is below a certain threshold, the execution logs are fed to the agent (i.e., logs are replayed) to adapt to the new changes. Both steps *Setup* and *Adapt* are done offline (i.e., are not done during a build process) via replaying the test cases execution logs, and therefore none of them delays the CI build. However, the ranking by the RL agent and calculation of test case features can delay the CI build. As we will discuss in Section 5, the ranking time is negligible and the calculation time of test case features is in the order of seconds.

5 VALIDATION

This section reports on the experiments we conducted to assess the accuracy and cost of the proposed RL configurations, and compare them with baselines. We first discuss the datasets of the study, evaluation metrics, comparison baselines, research questions, and our experimental setup. Then, we present the results and discuss their practical implications. The source code of our implementation and the results of experiments can be found here.[2]

5.1 Datasets

We ran experiments on two categories of datasets: simple and enriched history datasets. The former consists of the execution history of two projects that were made publicly available by previous work [15]. As discussed in Definition 2, simple history data only contains the age, average execution time, and verdicts of test cases. Such datasets are representative of regression testing situations where source code is not available. Enriched datasets (six projects) consist of the augmented history data (execution history and code features) from the Apache Commons projects, which were made publicly accessible [6]. The projects are written in Java and their build is managed using Maven. Enriched datasets represent testing situations where source code is available but full coverage analysis is not possible, due to the time constraints imposed by CI.

Table 3 lists the characteristics of the datasets. They contain the execution logs of 55-to-438 CI cycles, each of which contains at least six test cases. We do not consider cycles with less test cases as (1) there is no benefit to applying prioritization on a few test cases, and, (2) more importantly, as we will discuss later (Section 5.2), ranking a few test cases is not a challenging task and tends to inflate the accuracy results, as even random ranking can be suitable. The number of test case execution logs ranges from 2,207 to 32,118.

TABLE 3
Data Sets

Data set	Type.	Cycles	Logs	Fail Rate (%)	Failed Cycles	Avg. Calc. Time (Avg) Enriched Features [6]
Paint-Control	Simple	332	25,568	19.36	252	NA
IOFROL	Simple	209	32,118	28.66%	203	NA
Codec	Enriched	178	2,207	0%	0	1.78
Compress	Enriched	438	10,335	0.06%	7	3.64
Imaging	Enriched	147	4,482	0.04%	2	5.60
IO	Enriched	176	4,985	0.06%	3	2.88
Lang	Enriched	301	10,884	0.01%	2	5.58
Math	Enriched	55	3,822	0.01%	7	9.46

Further, the failure rates and the number of failed cycles (i.e., cycles that failed due to the failure of at least one regression test case, as defined in Definition 1) in enriched datasets are very low, ranging from 0 to 0.06 and 0 to 7, respectively, while the failure rates and the number of failed cycles in simple datasets are abnormally high, ranging from 19.36 to 28.43 percent and 203 to 252, respectively.

Finally, the last column of Table 3, shows the average calculation time of enriched features per cycle (Definition 2), based on the paper [6] that shared the enriched datasets. The calculation time ranges between 1.78 and 9.46 seconds per cycle across all datasets, which we consider a reasonable overhead in practice.

5.2 Evaluation Metrics

We use two evaluation metrics to measure the accuracy of prioritization techniques, that are both used in the literature and are described below in turn.

5.2.1 Normalized Rank Percentile Average (NRPA)

We adopt the Normalized Rank Percentile Average (NRPA) [6] for two reasons: (1) its capacity to measure the overall performance of a ranking, regardless of the context of the problem or the ranking criteria (e.g., fault detection for test prioritization), and (2) to be able to compare with the related work that uses *NRPA* as the evaluation metric. *NRPA* measures how close a predicted ranking of items is to the optimal ranking, i.e., the proportion of the optimal ranking that is contained in the predicted one. *NRPA* ranges from 0 to 1, where higher values are preferable. Assuming a ranking algorithm R that takes a set of k items and generates an ordered sequence s_e, then $NRPA = \frac{RPA(s_e)}{RPA(s_0)}$, where s_o is the optimal ranking of the items. Given any order of items (sequence s),

$$RPA(s) = \frac{\sum_{m \in s} \sum_{i=idx(s,m)}^{k} |s| - idx(s_o, m) + 1}{k^2(k+1)/2},$$

where $idx(s, m)$ returns the position of m in sequence s, and the lowest rank is given to an item with the highest priority.

5.2.2 Average Percentage of Faults Detected (APFD)

While *NRPA* is a suitable metric for measuring the accuracy of a ranking independently of the context, for regression testing, ranking failed test cases correctly (assigning them
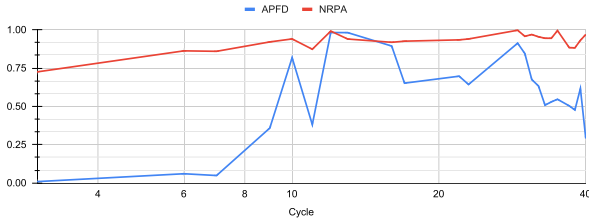
Fig. 4. APFD versus NRPA for algorithm A2C across the first 40 cycles of dataset paint-control.

Fig. 5. Worst and average NRPA values for 100 random rankings, across a range from 1 to 40 test cases (logarithmic scale).

the highest priority) is much more important than ranking the rest of the test cases. However, *NRPA* treats each test case equally regardless of their verdict and can be a misleading metric for test case prioritization in the presence of failures. For example, as shown in Fig. 4 depicting the performance of the A2C algorithm using the pairwise model, for the *Paint-Control* dataset in terms of APFD and NRPA. Algorithm A2C performs poorly in cycles 1-10, 13, and 40 in terms of prioritizing failed test cases, which is captured correctly by lower APFD values for these cycles. However, NRPA values contradict APFD values in all of these cycles (especially in cycle 40) because passing test cases are ranked properly based on their execution time and, therefore, they have high NRPA values. We observed such contradictory patterns in all of the datasets, thus suggesting that NRPA is not a good metric in the presence of failures, especially when only a small percentage of test cases fail per cycle. Therefore, we also make use of the well-known *APFD* metric, since it measures how well a certain ranking can reveal faults early.

APFD measures the weighted average of the percentage of faults detected by the execution of test cases in a certain order. It ranges from 0 to 1, with higher numbers implying faster fault detection. The *APFD* of an order s_e is calculated as

$$APFD(s_e) = 1 - \frac{\sum_{t \in s_e} idx(s_e, t) * t.v}{|s_e| * m} + \frac{1}{2 * |s_e|},$$

where m refers to the total number of faults.

While reviewing related work, we observed anomalies in the way APFD and NRPA were used and interpreted: (1) they assumed an APFD value of 1 even when there were no failed test cases in a cycle, which led to misleading results, especially when the work reported the average APFD across all cycles, and (2) NRPA is reported for cycles with only a few test cases, even for cycles with one test case that always results in NRPA=1. To remedy these issues, (1) when there is no failed test case in a cycle, we do not report APFD and use NRPA as an alternative, and (2) we ignore all cycles with less than six test cases and do not report NRPA or APFD for them. To further justify our choice, in Fig. 5, we report the worst and average NRPA values obtained with random ranking for a range of items. For five or less, these values are high (>.60 and >.80, respectively) and, therefore, including such NRPA values in the evaluation would unrealistically boost the results.

Also, it is worth mentioning that while reaching the optimal ranking is challenging (NRPA = 1), finding the worst ranking, in which no test case is ranked correctly (NRPA = 0), is also difficult. Therefore, the value of
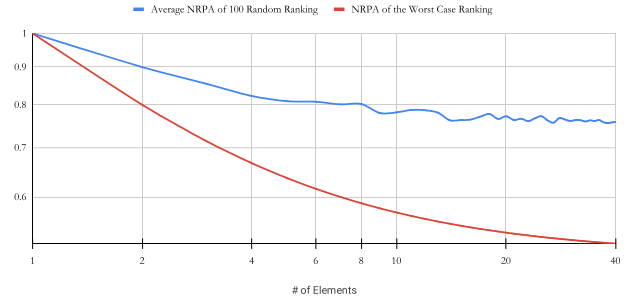
NRPA can be relatively high, even for random ranking, and needs to be interpreted carefully.

5.3 Comparison Baselines

Two papers report the use of RL for test case prioritization in the context of CI. The first [15] (RL-BS1) applies RL on three simple history datasets. Since the source code and data was made publicly available, we use this work as the first RL baseline (RL-BS1) and compare our suggested RL strategies with the best RL configuration from that work, based on the two out of three simple datasets used by the paper (the first two datasets in Table 3). We left one dataset out, the Google Shared dataset of Test suite Results (GSDTR) that was originally provided by Elbaum *et al.* [46]. GSDTR contains a sample of 3.5 Million test suite execution results from a fast and large scale continuous testing infrastructure of several google products, developed using different programming languages. Unfortunately, RL-BS1 treats the entire dataset as if it were the CI execution logs of one product, which is not correct. Since the main focus, in most ML-based test case prioritization work, including this work and RL-BS1, is the creation of a ML model for a specific product based on its CI logs, using this dataset is not an option. Note that we have tried to divide the dataset into separate product-specific datasets but product information is missing.

Second, the most recent related work by Bertolino *et al.* [6] applies three different implementations (i.e., Shallow Network, Deep Neural Network, Random Forest) of RL on enriched datasets (the last six datasets). The datasets, implementations, and detailed results of these experiments were also made publicly available and we also use the best configuration of this work as a second baseline (RL-BS2) to compare our work based on the six enriched datasets.

Note, however, that our reported results deviate from published results due to the issues we considered to be inappropriate usage of APFD and NRPA, as discussed in Section 5.2. Further, the study included cycles with less than five test cases, as discussed in Section 5.2, thus matching situations where prioritization is not needed and also resulting in inflated average APFD and NRPA values. Last, we applied each of the baselines only on the datasets for which they were originally used, because their implementation does not support the use of other datasets, thus requiring changes which might introduce errors.

While the primary motivation of the use of RL is in dealing with the dynamic nature of test prioritization in a CI context, we still would like to compare RL performance with the best supervised learning technique. We would like to determine whether we can benefit from the practical advantages of RL (adaptation) without losing significant accuracy in ranking compared with supervised learning, as reported by Bertolino *et al.* [6]. Thus, we use the best ranking supervised learning technique (MART), based on reported results, as the third baseline (MART) of comparison. As discussed in Section 1, MART does not support incremental learning [7], a feature that is essential to cope with frequently-changing CI environments.

5.4 Research Questions

- *RQ1* How do the selected state-of-the-art RL solutions (Table 2) perform in terms of prioritization accuracy and cost, using the simple and enriched datasets?
    - *RQ1.1* Which combinations of RL algorithms and ranking models perform better?
    - *RQ1.2* Which of the three ranking models (pointwise, pairwise, and listwise) perform better across RL algorithms?
    - *RQ1.3* Which of the RL algorithms perform better across ranking models?
- *RQ2* How does the best RL solution identified in RQ1 perform w.r.t the comparison baselines, based on their respective original datasets (Section 5.3)?
    - *RQ1.1* How does the best RL solution perform compared to the RL baselines (RL-BS)?
    - *RQ2.2* How does the best RL solution perform compared to the ML baseline (MART)?

5.5 Experimental Setup and Configurations

There are three ranking models from information retrieval. Thus to make our study comprehensive, we implemented[3] the three ranking models using state-of-the-art RL algorithms, as discussed in Section 4.2. We have used the Gym library [65] to simulate the CI environment using execution logs and relied on the implementation of RL algorithms provided by the Stable Baselines (v2.10.0) [12]. For each dataset, we ran three experiments corresponding to the three ranking models: pairwise, listwise, and pointwise. As discussed in Section 4.2, the applicability of RL algorithms is limited by the type of their action space. Thus, pairwise and listwise models involve seven experiments for each dataset, one per each RL technique that can support discrete action spaces (i.e., A2C, TRPO, PPO1-2, ACER, ACKTR, and DQN). Similarly, pointwise involve eight experiments for each dataset using RL techniques that support continuous action spaces (i.e., A2C, TRPO, PPO1-2, SAC, DDPG, ACKTR, and TD3). During the experiments, we observed that training using ACKTR with listwise ranking is extremely slow (more than 12 hours for a cycle), which makes it inappropriate for this work. Thus we dropped experiments using ACKTR and listwise ranking. The total number of experiments is therefore 168, during each of which an agent is trained for each

cycle, and then tested on the subsequent cycles. Overall, this resulted in 308,448 (number of all cycles * 168 experiments) RL agent training and evaluation instances. The process of agent training and evaluation is incremental as it is started by training an agent by replaying the execution logs of the first cycle, followed by the evaluation of the trained agent on the second cycle, which is then followed by replaying logs of the second cycle to improve the agent, and so on.

To ensure that enough training takes place, we used the minimum of $200 * n * log_2(n)$ (corresponding to 200 episodes for each pairwise training instance) and one million steps for each training instance (training for each cycle), where n refers to the number of test cases of the cycle. We stop when we either reach the set budget of steps per training instance or when the episode reward (sum of rewards across the steps of an episode) cannot be improved for more than 100 consecutive episodes (i.e., when the agent reaches plateau). The number of steps above is set based on the worst-case scenario, i.e., the pairwise training, in which an episode of training requires $n * log_2(n)$ steps. Often, RL algorithms come with a set of parameters that can be tuned. However, we use default parameters for all the experiments and leave parameter optimization to future experiments. Each experiment was run once using the HPC facilities of the University of Luxembourg [66] with the same configuration of 3 CPU cores, and 20 GiB memory. During the experiments, the rank of each test, along with the required time for training and evaluation, were recorded to answer the RQs. Note that, even though each of the experiments is run once due to the massive computation time (more than 46 days with three CPU cores, and 20 GiB memory) required by all the experiments, our analysis is based on many cycles (see Table 3) and 308K training and evaluation instances, which allows us to account for randomness in RL algorithms and draw safe conclusions.

5.6 Results and Discussion

5.6.1 RQ1

Overview. Table 4 shows the averages and standard deviations of APFD and NRPA for the eight datasets, using different configurations (combinations of ranking model and RL algorithm). Each column and row corresponds to a dataset and configuration, respectively. For example, the first column reports on how different configurations perform with *Paint-Control*, and the first row shows how the combination of pointwise ranking and A2C works for all datasets. We use convention $[ranking\ model] - [RL\ algorithm]$ to refer to configurations in the rest of the paper. For example, *pairwise-A2C* refers to a configuration of the pairwise ranking model and A2C algorithm.

Definition 5 (Relative Performance Rank). *For each dataset (column), the relative performance ranks of configurations in terms of APFD or NRPA are depicted with* (ⁿ)*, where a lower rank n indicates better performance. Assuming that* cnt *of a configuration* cf *denotes the number of configurations with a significantly lower average in terms of APFD or NRPA, a configuration with a higher* cnt *is ranked lower, and two configurations with identical* cnt *are ranked equal. For instance, for dataset IMAGE, configurations* pairwise-A2C *and* pairwise-ACER *are ranked*

TABLE 4
The Average Performance of Different Configurations in Terms of APFD and NRPA, along with the Results of the Three Baselines
(Section 5.3)

	RM	IOFROL (APFD)	Paint. (APFD)	CODEC (NRPA)	IMAG (NRPA)	IO (NRPA)	COMP (NRPA)	LANG (NRPA)	MATH (NRPA)
A2C	PA	.55±.13 ④	.72±.24 ①	.97±.04 ①	.96±.05 ①	.98±.02 ①	.98±.03 ②	.95±.04 ②	.96±.04 ④
	PO	.52±.14 ⑤	.57±.23 ⑤	.89±.07 ⑥	.92±.05 ⑧	.91±.05 ③	.92±.05 ⑥	.86±.07 ⑤	.90±.04 ②
	LI	.50±.10 ⑨	.48±.18 ⑨	.78±.08 ⑧	.77±.06 ⑫	.76±.05 ④	.76±.06 ⑨	.77±.05 ⑧	.76±.06 ⑤
ACER	PA	.56±.14 ③	.73±.22 ①	.98±.03 ①	.96±.06 ①	.98±.02 ①	.98±.02 ①	.96±.03 ①	.96±.04 ①
	LI	.50±.10 ⑨	.48±.19 ⑨	.79±.07 ⑧	.77±.06 ⑫	.76±.05 ④	.77±.05 ⑨	.77±.05 ⑧	.77±.05 ⑤
ACKTR	PA	.57±.13 ①	.68±.22 ②	.93±.09 ⑤	.94±.07 ⑤	.77±.05 ④	.97±.03 ④	.95±.04 ③	.95±.05 ①
	PO	.52±.14 ⑤	.57±.24 ⑤	.77±.08 ⑧	.78±.08 ⑪	.74±.08 ④	.77±.06 ⑨	.78±.06 ⑦	.78±.06 ⑤
DDPG	PO	.52±.13 ⑤	.62±.22 ④	.88±.08 ⑦	.82±.07 ⑩	.87±.09 ③	.82±.07 ⑧	.80±.07 ⑥	.86±.07 ③
DQN	PA	.53±.13 ⑤	.67±.23 ③	.94±.06 ③	.95±.06 ⑤	.98±.02 ①	.97±.03 ③	.95±.04 ②	.94±.05 ①
	LI	.50±.10 ⑧	.50±.19 ⑧	.79±.07 ⑧	.77±.06 ⑫	.76±.05 ④	.77±.05 ⑨	.77±.05 ⑧	.76±.05 ⑤
PPO1	PA	.56±.14 ③	.72±.24 ①	.97±.04 ②	.96±.05 ③	.98±.02 ①	.98±.03 ②	.95±.04 ②	.96±.04 ①
	PO	.52±.14 ⑤	.58±.24 ⑤	.89±.09 ⑥	.93±.05 ⑦	.90±.05 ③	.90±.05 ⑦	.86±.06 ⑤	.84±.07 ④
	LI	.51±.11 ⑦	.56±.23 ⑦	.79±.07 ⑧	.77±.05 ⑫	.76±.06 ④	.77±.06 ⑨	.78±.06 ⑦	.78±.06 ⑤
PPO2	PA	.57±.13 ②	.71±.23 ②	.97±.04 ①	.96±.05 ②	.98±.02 ①	.98±.02 ②	.96±.03 ②	.96±.04 ①
	PO	.52±.14 ⑤	.57±.24 ⑤	.93±.06 ④	.93±.05 ⑥	.95±.04 ②	.94±.04 ⑤	.89±.05 ④	.85±.06 ④
	LI	.51±.10 ⑥	.49±.22 ⑧	.79±.08 ⑧	.78±.06 ⑪	.76±.06 ④	.77±.05 ⑨	.78±.06 ⑧	.77±.07 ⑤
SAC	PO	.52±.14 ⑤	.57±.24 ⑥	.78±.09 ⑧	.76±.07 ⑫	.75±.08 ⑨	.76±.08 ⑨	.77±.07 ⑧	.79±.07 ⑤
TD3	PO	.52±.14 ⑤	.58±.24 ⑤	.78±.09 ⑧	.78±.07 ⑪	.75±.08 ④	.77±.07 ⑨	.77±.07 ⑧	.76±.07 ⑤
TRPO	PA	.57±.13 ②	.71±.24 ①	.96±.04 ②	.95±.07 ④	.98±.03 ①	.98±.02 ②	.95±.04 ②	.95±.05 ①
	PO	.52±.14 ⑤	.57±.23 ⑤	.90±.07 ⑥	.92±.05 ⑨	.94±.04 ②	.92±.04 ⑥	.90±.06 ④	.86±.06 ③
	LI	.50±.11 ⑧	.48±.19 ⑨	.80±.08 ⑧	.76±.05 ⑫	.77±.07 ④	.76±.06 ⑨	.77±.06 ⑧	.78±.06 ⑤
Optimal	NA	.79±.14	.89±.14	NA	NA	NA	NA	NA	NA
RL-BS1	PO	.63±.16	.74±.24	NA	NA	NA	NA	NA	NA
RL-BS2	PO	NA	NA	.90±.05	.89±.09	.84±.13	.90±.05	.89±.07	.84±.13
MART	PR	NA	NA	.96±.03	.90±.05	.93±.02	.96±.02	.94±.04	.95±.02

PR: *Pairwise*, PO: *Pointwise*, LI: *Listwise*
The index in each cell shows the position of a configuration (row) with respect to others for each dataset (column) in terms of NRPA or APFD, based on statistical testing.

①, *because the cnt of all three configurations are equal and significantly higher than other configurations.*

To check for significant differences in results across configurations for a given dataset, we use Welch's ANOVA [67] to compare all configurations across all cycles, with one NRPA or APFD value per cycle. Then we perform the Games-Howell post-hoc test [68] to compare each pair of configurations. The significance level is set to 0.05, and therefore any difference with p-value $< = 0.05$ is considered significant. We use Welch's ANOVA rather than one-way ANOVA because the variances in results across configurations are not equal. Also, we rely on the Games-Howell post-hoc test due to its compatibility with Welch's ANOVA (i.e., no assumption about equal variance). Using this post-hoc test also addresses the usual problems related to repeated testing when performing multiple comparisons (increased type-I error) [69].

Table 6 shows the overall training times for all experiments. Similarly, Table 7 shows the averages and standard deviations of prediction (ranking) time for all experiments. Each column and row corresponds to a dataset and configuration, respectively. For each dataset (column), the relative performance ranking of configurations in terms of training/ prediction time are depicted with $\textcircled{t_i}$ where a lower rank n indicates shorter training/prediction time. The relative performance ranks are calculated according to Definition 5, but based on training/prediction time.

RQ1.1 As shown in Table 4, multiple pairwise configurations perform best for some of the datasets but we cannot single out one configuration overall based on ranks. Configuration pairwise-ACER yields, however, the best averages. Also, based on the post-hoc test, pairwise-ACER perform best across all datasets except one, followed by pairwise-A2C and pairwise-PPO2. In contrast, listwise ranking with all algorithms, pointwise-TD3, and pointwise-SAC perform worst. As discussed in Section 4.1.1, we argue that due to the large observation space of listwise, it requires extensive training data, and under the same circumstances (i.e., same training data and steps of training) their accuracy can not be as good as the pairwise and pointwise configurations.

Also, for each dataset, we measured the effect size of the differences between configuration pairs based on Common Language Effect Size (CLE) [70], [71]. CLE estimates the probability that a randomly sampled score from one population will be greater than a randomly sampled score from

TABLE 5
Common Language Effect Size Between One of the Worst and
Best Configurations for Each Data Set Based on Accuracy

Data set	Best Conf.	Worst Conf.	CLE
IOFROL	PAIRWISE-ACKTR	LISTWISE-ACER	.701
Paint.	PAIRWISE-A2C	LISTWISE-TRPO	.786
CODEC	PAIRWISE-PPO2	LISTWISE-TRPO	.973
IMAG	PAIRWISE-A2C	LISTWISE-TRPO	.981
IO	PAIRWISE-DQN	LISTWISE-TRPO	.999
COMP	PAIRWISE-ACER	LISTWISE-TRPO	.997
LANG	PAIRWISE-ACER	LISTWISE-TRPO	.986
MATH	PAIRWISE-DQN	LISTWISE-TRPO	.963

the other population. As shown in Table 5, CLE values between one of the worst and best cases for the six enriched datasets are above 96 percent, while they are 79 and 70 percent for the simple datasets Paint-Control and IOFROL, respectively. Results therefore show the importance of selecting one of the best configurations. Also, relatively smaller CLEs for simple datasets suggests none of the configurations learned an adequate ranking strategy in these cases. This may not be surprising since learning an accurate policy for complex software systems cannot be expected to be always possible based on simple data.

In terms of training time, as shown in Table 6, based once again on the post-hoc test, multiple configurations (pairwise or pointwise) perform well for some of the datasets. Still, we cannot recommend one specific configuration overall. Pairwise-TRPO and pairwise-PPO1 are the most efficient in terms of training time, followed by pairwise-ACER. In

contrast, listwise and pairwise rankings with the DQN algorithm feature the worst training time. Also, pointwise-DDPG, pointwise-TD3, and pointwise-SAC are relatively slow across all datasets. As discussed in Section 4.3, since our approach uses offline training for the initial setup and adapting to new changes, training can therefore occur in the background without adding any delay to the CI build process. Thus, the differences in training time across configurations, which are in the order of minutes, do not constitute a practical issue. Also, the training in pointwise and pairwise modes could be made more efficient by relying on sampling to only replay a random and small subset of test execution logs during training.

In terms of prediction time, as shown in Table 7, based once again on the post-hoc test, multiple pointwise configurations perform well for some of the datasets. Still, we cannot recommend one specific configuration overall. Pointwise-PPO1 and pointwise-A2C are the most efficient in terms of prediction time, which are followed by pointwise-A2C. In contrast, listwise ranking with the ACER, PPO1, PPO2, and TRPO algorithms feature the worst prediction times. The prediction times of pointwise and pairwise configurations do not exceed 2.22 seconds across all datasets and are therefore negligible overheads for CI builds. Thus, considering the low data collection time for enriched datasets (<10 seconds, last column of Table 3), it is safe to conclude that pairwise and pointwise approaches are practical choices in terms of computation overhead. In contrast, in several cases, prediction times of listwise configurations spike to more than 200 seconds and may not be

TABLE 6
The Sum of Training Time (Minutes) for all Cycles Across Datasets and Configurations

	RM	IOFROL	Paint.	CODEC	IMAG	IO	COMP	LANG	MATH
A2C	PA	911.8 ②	713.5 ③	187.4 ②	213.0 ⑨	253.2 ⑩	986.0 ④	602.3 ④	107.4 ④
	PO	1047.7 ③	737.5 ③	227.0 ⑥	204.9 ⑧	237.3 ⑨	1724.4 ⑮	1026.0 ⑪	149.3 ④
	LI	1718.3 ⑤	775.1 ④	209.1 ⑤	181.0 ⑦	26.3 ⑪	1142.8 ⑩	545.4 ③	222.9 ④
ACER	PA	944.6 ③	715.6 ③	175.7 ①	147.3 ③	184.8 ③	905.4 ①	61.2 ④	11.9 ④
	LI	1983.7 ⑥	74.7 ③	223.4 ⑥	155.7 ④	196.4 ④	101.2 ⑤	71.7 ⑦	254.0 ④
ACKTR	PA	949.8 ③	762.1 ④	189.9 ②	161.7 ⑤	199.1 ④	1012.9 ⑤	635.1 ⑤	127.3 ④
	PO	1032.4 ③	641.7 ②	202.6 ④	148.5 ③	231.1 ⑨	109.6 ⑧	665.4 ⑥	122.2 ④
DDPG	PO	2576.0 ⑦	1509.5 ⑦	254.9 ⑨	254.4 ⑪	349.6 ⑫	1133.3 ⑩	985.5 ⑩	221.6 ④
DQN	PA	2021.0 ⑥	1343.4 ⑥	242.3 ⑧	389.2 ⑬	429.0 ⑬	1277.8 ⑬	91.7 ⑩	236.5 ④
	LI	10652.9 ⑩	3602.9 ⑩	638.1 ⑫	564.8 ⑭	629.3 ⑮	2049.1 ⑯	2024.2 ⑭	1842.2 ⑧
PPO1	PA	848.9 ①	664.1 ②	17.6 ①	133.8 ②	167.8 ①	908.8 ①	555.6 ③	98.3 ③
	PO	836.4 ①	668.5 ②	186.5 ②	138.9 ③	203.9 ⑥	1102.9 ⑨	845.4 ⑨	72.4 ①
	LI	5183.4 ⑨	1194.5 ⑥	226.9 ⑥	225.3 ⑩	273.5 ⑪	1201.5 ⑫	961.5 ⑩	713.4 ⑦
PPO2	PA	922.8 ②	72.1 ③	316.5 ⑪	174.7 ⑥	224.0 ⑧	961.3 ③	607.5 ④	107.5 ④
	PO	994.3 ③	623.8 ②	219.8 ⑥	149.7 ③	231.9 ⑨	878.2 ①	675.0 ⑥	86.2 ②
	LI	1726.0 ④	731.5 ③	195.0 ③	165.1 ⑤	243.8 ⑨	1027.3 ⑥	524.1 ②	197.7 ④
SAC	PO	3413.4 ⑦	259.3 ⑨	336.8 ⑪	474.2 ⑬	483.7 ⑭	1592.4 ⑮	1269.6 ⑬	455.7 ⑥
TD3	PO	3592.1 ⑧	1821.2 ⑧	291.3 ⑩	37.1 ⑫	414.3 ⑬	1318.9 ⑭	1086.5 ⑫	333.8 ⑤
TRPO	PA	758.3 ①	611.1 ①	189.5 ②	13.5 ①	176.4 ②	927.6 ②	556.8 ③	89.4 ②
	PO	806.9 ①	626.3 ②	207.2 ⑤	135.0 ②	201.6 ⑤	1065.8 ⑦	481.5 ①	66.6 ①
	LI	3005.5 ⑦	855.5 ⑤	217.6 ⑦	161.1 ⑤	215.1 ⑦	1151.1 ⑪	82.6 ⑧	371.5 ⑤

PR: *Pairwise*, PO: *Pointwise*, LI: *Listwise*.

TABLE 7
The Average of Prediction (ranking) Time (Seconds) for all Cycles Across Datasets and Configurations

	RM	IOFROL	Paint.	CODEC	IMAG	IO	COMP	LANG	MATH
A2C	PA	1.3±0.9⑦	1.0±0.2④	0.8±0.1①	0.8±0.1⑤	0.7±0.1①	0.8±0.1④	0.9±0.1③	1.1±0.5⑥
	PO	1.0±0.1③	0.8±0.1②	0.9±0.1④	0.6±0.1①	0.7±0.1①	0.6±0.0①	0.6±0.0①	0.7±0.1②
	LI	32±142⑥	4±14⑦	0.9±0.4③	1.4±2.9⑧	2.0±3.5⑨	4±25⑧	1.2±0.6⑧	5±20⑦
ACER	PA	2±0.9⑭	1.9±0.2⑩	1.8±0.1⑭	1.4±0.1⑬	1.4±0.1⑫	1.6±0.1⑭	1.8±0.1⑭	2.0±0.5⑬
	LI	11±26⑮	3±2⑫	2±0.2⑮	1.7±0.3⑮	1.7±0.4⑬	2.0±0.8⑯	3±1.3⑮	4±3⑭
ACKTR	PA	1.6±0.8⑪	1.4±0.2⑧	1.2±0.1⑦	1.0±0.1⑨	1.0±0.1④	1.2±0.1⑨	1.2±0.1⑩	1.4±0.5⑪
	PO	1.2±0.2⑤	1.0±0.1④	1.3±0.1⑨	1.0±0.1⑨	1.1±0.1⑦	1.2±0.1⑪	1.3±0.1⑪	1.2±0.1⑨
DDPG	PO	0.9±0.2②	0.9±0.1③	0.9±0.1②	0.7±0.1②	0.7±0.1①	0.7±0.1②	0.8±0.1②	0.7±0.2②
DQN	PA	1.7±1.0⑫	1.3±0.3⑧	1.2±0.3⑦	1.0±0.3⑩	1.0±0.3④	1.2±0.3⑩	1.2±0.2⑨	1.3±0.4⑩
	LI	9±44⑥	2.0±0.3⑪	1.5±0.6⑪	1.2±0.3⑪	1.1±0.2⑦	1.1±0.3⑩	1.3±0.4⑪	2±0.7⑮
PPO1	PA	1.4±1.0⑧	1.0±0.3⑤	0.8±0.2②	0.7±0.1③	0.7±0.2①	0.8±0.3⑤	0.9±0.2④	1.1±0.5⑥
	PO	0.8±0.1①	0.8±0.1②	0.8±0.2①	0.7±0.2③	0.7±0.1①	0.8±0.2④	0.7±0.2①	0.6±0.1①
	LI	406±405⑯	329±234⑮	3±6⑬	4±7⑯	19±72⑯	8±38⑧	62±173⑯	50±181⑦
PPO2	PA	1.5±1.0⑩	1.1±0.2⑥	0.9±0.1⑤	0.9±0.1⑦	1.0±0.1③	1.0±0.1⑦	1.0±0.1⑥	1.2±0.5⑧
	PO	1.0±0.1④	0.8±0.1①	1.0±0.1⑥	0.7±0.1④	0.9±0.1②	0.7±0.1③	0.9±0.1⑤	0.7±0.1③
	LI	396±462⑯	217±254⑭	4±24⑦	87±193⑰	14±47⑧	8±38⑧	32±104⑯	44±137⑦
SAC	PO	1.5±0.1⑨	1.2±0.1⑦	1.5±0.1⑫	1.2±0.1⑫	1.3±0.1⑩	1.2±0.1⑨	1.5±0.1⑬	1.1±0.1⑤
TD3	PO	1.0±0.1④	1.2±0.1⑦	1.2±0.1⑦	0.9±0.1⑥	1.0±0.1⑤	0.9±0.1⑥	1.1±0.1⑦	0.9±0.1④
TRPO	PA	1.9±0.9⑬	1.5±0.4⑨	1.4±0.3⑩	1.2±0.2⑪	1.2±0.4⑦	1.4±0.4⑫	1.4±0.3⑫	1.6±0.5⑫
	PO	1.3±0.3⑥	1.6±0.4⑨	1.6±0.5⑪	1.2±0.3⑫	1.4±0.3⑪	1.5±0.5⑬	1.2±0.2⑩	1.2±0.1⑨
	LI	245±409⑯	47±118⑬	1.2±0.3⑧	1.8±1.3⑭	2±4⑨	1.9±1.6⑮	158±287⑰	11±21⑦

PR: *Pairwise*, PO: *Pointwise*, LI: *Listwise; Values greater than 2 are rounded.*

applicable in practice. A possible reason is related to how listwise ranking is modeled in Algorithm 2, in which dummy test cases are used for padding to prevent test cases from being selected repeatedly. Thus, when the agent cannot learn a suitable policy, it selects dummy test cases many times, and that increases prediction time.

RQ1.2 As discussed above, pairwise ranking configurations fare relatively better than pointwise and listwise ranking in terms of accuracy (NRPA and APFD). Since applicable algorithms differ for each ranking model, such superior performance may therefore be due to the RL algorithms rather than the pairwise ranking model. Thus, to investigate this hypothesis, we perform four sets of Welch ANOVA and Games-Howell post-hoc tests based on the results of each of the four algorithms (A2C, PPO1, PPO2, TRPO) shared across ranking models and all enriched datasets. For each such algorithm, we create three sample groups corresponding to three ranking models based on the results of the algorithms for enriched datasets. We repeat the same analysis for the simple datasets too. The results show that, for enriched and simple datasets, there is a statistically significant difference for each algorithm across ranking models. Further, in all cases, regardless of the algorithm, pairwise fares better than pointwise and listwise. Similarly, pointwise fares better than listwise in all cases, except for PPO1 where they are comparable.

As discussed above (RQ1.1), due to the large observation space, the accuracy of listwise configurations is relatively lower than pointwise and pairwise. We also conjecture that in pairwise configurations, using a pair of test cases allows the agent to be trained on richer feature sets (see

Section 4.1.3) compared to the pointwise configurations that use a set of features based on a single test case (see Section 4.1.2). In general, a ranking model that is trained based on pointwise features can be coarse due to the limited information captured by single document features [72]. This is why state-of-the-art ranking models tend to use pairwise ranking [58] (e.g., RankNet, LambdaRank, and LambdaMART).

Concerning training time, again, we performed four sets of Welch ANOVA and Games-Howell post-hoc test for enriched and simple datasets, as discussed above but based on training time. The results show that, for enriched datasets, there is a statistically significant difference for each algorithm across ranking models. Further, in all cases, regardless of the algorithm, listwise fares worse than pointwise and pairwise, except A2C for which pointwise fares worst. Also, pairwise fares better than other models except for TRPO for which pairwise and pointwise are similar. Similarly, for simple datasets, listwise is worse than other models. Pairwise and pointwise have similar training time in all cases. As discussed in RQ1.1, however, the differences in training time across configurations, which are in the order of minutes, do not constitute a practical issue in our context.

Again, we performed four sets of Welch ANOVA and Games-Howell post-hoc tests for enriched and simple datasets, as discussed above but based on prediction time. The results show that there is a statistically significant difference for each algorithm across ranking models, for both enriched and simple datasets. Further, in all cases, regardless of the algorithm, listwise fares worse than pointwise and pairwise.

Also, pointwise fares better than pairwise. As discussed in RQ1.1, the high prediction time of listwise ranking can entail practical issues in our context. However, the differences in prediction time between pairwise and pointwise across configurations are less than 2.22 seconds and do not have practical implications.

RQ1.3 To analyze the relative accuracy of RL algorithms, we perform three sets of Welch ANOVA and Games-Howell post-hoc tests corresponding to the three ranking models, based on the result of all algorithms, across enriched datasets. Then we repeat the same analysis for simple datasets. The result shows that there are no significant differences between RL algorithms for enriched datasets when using the listwise ranking model. Similarly, there are no significant differences for simple datasets when using pointwise ranking. The results for the remaining cases are different as described next, assuming that where > and = denote greater and equal performance rank, respectively, as calculated in Definition 5.

- *Pairwise and enriched datasets:* ACER > PPO2 = A2C = PPO1 > TRPO > DQN > ACKTR
- *Pointwise and enriched datasets:* PPO2 > TRPO > A2C > PPO1 > DDPG > SAC = TD3 = ACKTR.
- *Pairwise and simple datasets:* ACER > PPO2 = A2C = PPO1 > TRPO = ACKTR > DQN
- *Listwise and simple datasets:* PPO1 > PPO2 = A2C = PPO1 = TRPO = ACER = DQN

To compare training time, we perform an analysis similar to the one above but based on training time at each cycle. Results clearly show that there is a significant difference across the training time of algorithms using the same ranking models, as described next.

- *Pairwise and enriched datasets:* DQN > PPO2 > ACKTR = A2C > ACER > TRPO = PPO1
- *Pointwise and enriched datasets:* SAC > TD3 = A2C < DDPG > PPO1 > ACKTR > PPO2 > TRPO
- *Listwise and enriched datasets:* DQN > PPO1 > TRPO > PPO2 = A2C = ACER = DQN
- *Pairwise and simple datasets:* DQN > ACKTR > A2C = PPO1 = PPO2 = ACER > TRPO
- *Pointwise and simple datasets:* SAC = DDPG = TD3 > A2C = PPO1 = PPO2 = ACKTR = TRPO
- *Listwise and simple datasets:* DQN > PPO1 > TRPO > ACER = PPO2 = A2C

To compare prediction times, we perform an analysis similar to the one above at each cycle. Results clearly show that there is a significant difference across the prediction time of algorithms using the same ranking models, as described next.

- *Pairwise and enriched datasets:* A2C = PPO1 < PPO2 < DQN = ACKTR < TRPO < ACER
- *Pointwise and enriched datasets:* A2C < DDPG = PPO1 < PPO2 < TD3 < ACKTR < SAC < TRPO
- *Listwise and enriched datasets:* DQN < A2C < ACER < PPO1 < PPO2 < TRPO
- *Pairwise and simple datasets:* A2C < PPO1 < PPO2 < DQN = 'ACKTR' < TRPO < ACER
- *Pointwise and simple datasets:* PPO1 > DDPG > PPO2 > A2C: = TD3 > ACKTR = SAC > TRPO: 6

TABLE 8
Welch's t-test Results and Common Language Effect Size Between Pairwise-ACER and Baselines

	RL-BS1.		RL-BS2		MART	
	p-val	CLE	*p*-val	CLE	*p*-val	CLE
IO	NA	NA	.0000	.985	.0000	.931
CODEC	NA	NA	.0000	.942	.1057	.551
IMAG	NA	NA	.0000	.895	.0000	.854
COMP	NA	NA	.0000	.976	.0000	.795
LANG	NA	NA	.0000	.931	.0000	.677
MATH	NA	NA	.0000	.915	.1218	.681
Paint.	.7483	.486	NA	NA	NA	NA
IOFROL	.0000	.376	NA	NA	NA	NA

- *Listwise and simple datasets:* A2C = ACER < TRPO < PPO2 < PPO1

Based on the above results, we can conclude that: 1) DQN, SAC, and TD3 are the worst algorithms in terms of training time, and 2) listwise configurations feature the worst prediction times. Since the accuracy of listwise configurations, as mentioned earlier, is relatively low regardless of the algorithm, we can therefore recommend against their use for test case prioritization. In contrast, overall PPO2, A2C, and ACER are relatively fast to train, as their prediction time is less than 2.22 seconds for pairwise and pointwise. Further, since their accuracy, especially that of ACER, is relatively good using pairwise ranking, we can recommend using them with the pairwise model for test case prioritization.

Further, from the results of RQ1.3, we can see that one of the actor-critic algorithms always provides the best result in terms of accuracy, training time, and prediction time. Actor-critic algorithms combine the strong points of actor-only and critic-only methods. The critic estimates the value function, and the actor updates the policy distribution in the direction suggested by the critic (such as with policy gradients) [21]. Also, these algorithms, such as ACER and PPO2, use multiple workers to avoid the use of a replay buffer and this results in a faster training time [26].

5.6.2   RQ2

Overview. The averages and standard deviations of baselines for NRPA and APFD are shown in the last three rows of Table 4, for the datasets on which they were originally experimented. We used one of the best configurations (highest average accuracy) from RQ1 across all datasets (pairwise-ACER) and compared the results of this configuration with the baselines in terms of NRPA or APFD. As before, we used the Welch t-test since it does not assume equal variances. We calculate CLE again, this time between pairwise-ACER and baselines to assess the effect size of differences. Table 8 shows the results of Welch t-test and CLE for all datasets.

RQ2.1 Row RL-BS1 of Table 4 corresponds to the results of the RL-based solution proposed by Spieker *et al.* [15]. We have replicated their experiment and calculated APFD, since no detailed results were available online. As discussed in Section 5.3, this work originally relies on simple history data and, therefore, we did not attempt to apply it to
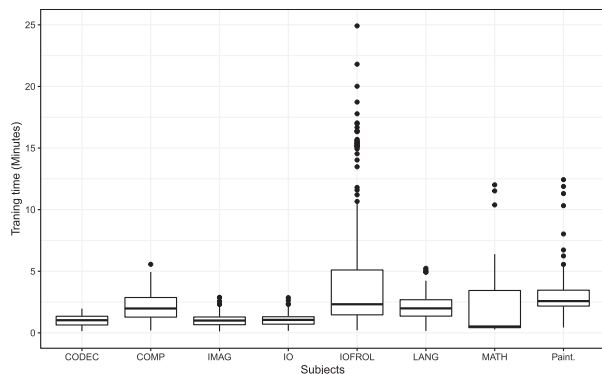
Fig. 6. Training time of pairwise-ACER for all datasets.

enriched datasets. As shown in the first two columns of row RL-BS1, for dataset Paint-Control, pairwise-ACER performs as well as RL-BS1, i.e., there are no statistically significant differences in results. Also, for dataset IOFROL, RL-BS1 fares slightly better than pairwise-ACER, however with a low CLE of 62.4. But overall, no approach (pairwise-ACER or RL-BS1) performs well, more particularly with IOFROL. It is interesting to note that the average APFD of RL-BS1 and pairwise-ACER (1) are around 0.15 lower than that of the optimal ranking (row *Optimal* of Table 4), and (2) are not significantly different from a simple heuristic solution that would prioritize test cases based on their recent verdicts with recently failed test cases assigned a higher priority (APFDs 0.632 and 0.772, for IOFROL and Paint-Control, respectively). These results suggest that simply relying on test execution history, such as the one available in simple datasets, does not provide sufficient features for learning an accurate test prioritization policy.

Row RL-BS2 shows the results of the best configuration (RL-BS2) among the RL-based solutions proposed by Bertolino *et al.* [6]. We have used the detailed experimental results available online to recalculate NRPA while ignoring the cycles with less than five test cases. As shown in the corresponding row, pairwise-ACER fares significantly better for all datasets. To check that the differences in average are statistically significant, we performed again a Welch t-test for each dataset. As shown in Table 8, pairwise-ACER is significantly better for all datasets. Further, CLE ranges between 0.89 to 0.98, which implies that for at least 89 percent of the cycles, pairwise-ACER performs better than RL-BS2.

Thus, according to the results above, we can safely conclude that pairwise-ACER significantly improves, in terms of ranking accuracy, the state-of-the-art regarding the use of RL for test case prioritization. Since the baselines use only one episode of training per cycle, their training time is significantly lower than our best configuration (pairwise-ACER), which is based on the pairwise ranking model. But as shown in Fig. 6, the average execution time per cycle is less than 5 minutes across all datasets and the worst-case training time is less 25 minutes. Since our approach uses offline training by replaying logs and enables the training process to be run in the background, such training times would not add any delay to the CI process. Therefore, the extra training time of pairwise-ACER compared to baselines, which is in the order of minutes, does not have practical consequences.

RQ2.2 Row MART (MART ranking model) in Table 4 provides the results of the best ML-based solution according to a comprehensive evaluation [6]. For all datasets, except CODEC and MATH where results are equivalent, pairwise-ACER performs better than MART. We once again performed a Welch t-test and, as shown in Table 8, the test results confirm that the differences for CODEC and MATH are not significant, while they are significant for the other four datasets. Also, to measure the effect size, we calculated CLE, as shown in Table 8. The CLE of pairwise-ACER versus MART ranges between 0.551 to 0.931 with an average of 0.75, i.e., in 75 percent of the cycles, pairwise-ACER fares better than MART. Therefore, we can safely conclude that pairwise-ACER advancthe state-of-the-art compared to the best ML-based ranking technique (MART).

In addition to their higher ranking accuracy over MART, RL agents can be incrementally trained to adapt to changes in the system and test suites, which is one of the main challenges in the context of frequently-changing and dynamic CI environments, in which new execution logs and code changes are continuously collected. In contrast, the MART ranking model is an ensemble model of boosted regression trees. Boosting algorithms, as a class of ensemble learning methods, are designed for static training, based on a fixed training set. Thus, they cannot be directly and easily applied to online learning and incremental learning [7]. Supporting incremental learning in boosting algorithms is an active research area for which no solution is currently available in existing libraries [73]. This causes practical issues since the performance of the ranking model will gradually decay after some cycles, and a new model needs to be trained based on the most recent data. In contrast, RL algorithms gradually adapt based on the incoming data and there is no need to train a new model from scratch.

As mentioned above, previous attempts to apply RL to test case prioritization had brought adaptability at the expense of accuracy. In this work, pairwise ranking using the ACER algorithm has significantly improved ranking accuracy over that of MART, the best reported ML-based ranking model for test case prioritization, as well as over that of previous attempts at using RL. Thus, we can conclude that the best RL configurations, for example based on a pairwise ranking model and the ACER algorithm, have the potential to be a reliable and adaptive solution for test case prioritization in CI environments.

5.7 Threats to Validity

The low and high failure rates of enriched and simple data sets, respectively, may threaten the generality of our evaluation results. However, we only use these data sets to make a fair comparison with related work under identical circumstances. We do not make any general claim on the effectiveness of our approach in absolute terms but rather focus on relative effectiveness across ranking models and RL techniques. Note that the fact that we get largely consistent results (in terms of trends, not magnitudes) with low and high failure rates is reassuring regarding the generalizability of the recommendations we provide regarding ranking models and RL algorithms.

Further, low failure rates such as the one for the enriched dataset tend to characterize the datasets typically used for evaluating test selection and prioritization techniques [74], [75]. Further, in a CI context, Beller *et al.* [76] conducted a comprehensive analysis of TravisCI projects and showed that for all 1,108 Java projects with test executions, the ratio of builds with at least one failed test case has a median of 2.9 percent and a mean of 10.3 percent. To deal with this issue, for evaluation purposes, studies focused on non-ML techniques rely on seeded faults, which are typically produced through hand-seeding or program mutation fault injection techniques [74], [75]. In the context of ML-based techniques, where the goal is to train an ML model based on the history of test executions and source code changes, using fault injection techniques is not a valid option since it would add some faults randomly into the system, that have no relation with history.

Another potential threat to validity is related to our evaluation metrics, which are standard across existing studies. We, however, in Section 5.2, discuss their limitations, how they should be interpreted, and when they should be used.

6 CONCLUSION

In this paper, we formalized and investigated test case prioritization in continuous integration (CI) environments as a Reinforcement Learning (RL) problem. Our main motivation is to benefit from the capacity of RL to seamlessly adapt to changes in systems and test suites, while potentially reaching high ranking accuracy of regression test cases. Such high accuracy would help detect as many regression faults as quickly as possible within the tight available resources that are typically available in CI contexts, where frequent changes take place.

Formalization is guided by the three well-known ranking models from the information retrieval domain: pairwise, pointwise, and listwise. Further, we have implemented this formalization by using a diverse set of carefully selected, state-of-the-art RL algorithms.

We then performed an extensive evaluation over eight subject systems by combining 10 RL algorithms with the three ranking models, that resulted in 21 RL configurations. The evaluation reveals that by using a pairwise ranking model and the ACER algorithm [27], an actor critic-based RL algorithm (pairwise-ACER), we obtain the best ranking accuracy. This accuracy, when enriching test execution history data with light-weight code features, is furthermore very close to the optimal ranking of test cases based on the actual failure data and execution times.

To position our work, we have compared pairwise-ACER with the two recent RL approaches and the best ML solution (MART) reported for test case prioritization. Using the standard NRPA ranking accuracy metric (ranging from 0 to 1), based on enriched datasets, the results show a significant ranking improvement when compared with both previous RL-based work (+0.1 on average) and MART (+0.027 on average). Further, we reach very high accuracy (NRPA >0.96), thus enabling the application of RL in practice. Simple datasets only based on execution history do not lead, with any learning technique, to satisfactory results. Differences in training time across approaches are not practically

relevant. Based on our results, we conclude that the use of our optimal RL configuration (pairwise-ACER) can provide, based on adequate history and code data, a reliable and adaptive solution for test case prioritization in CI environments.

While our work advances state-of-the-art in the use of RL techniques for test case prioritization, a certain number of issues remains open that should be tackled by future work. In the following, we discuss the most important ones.

- Tuning and optimization of our current approach. As discussed earlier, RL algorithms come with a set of hyperparameters that need to be tuned. However, we applied all of the algorithms with their default hyperparameters. Also, we only evaluated a limited number of reward functions that impact the performance of RL algorithms. Thus, optimization and tuning the best configuration of our approach (pairwise-ACER) is a natural next step to this work that can be performed automatically and systematically using search-based tuning frameworks such as Optuna [77].

- Preparing of a rich dataset and a benchmark. Though, for comparison purposes, we used existing datasets that were made available by previous studies, we also observed that the available datasets are limited in terms of features, the number of products, and the diversity of failure rates (failure rates in our benchmarks are either very low or very high). Also, there is no available benchmark for comparing available techniques based on a set of identical, representative datasets. As an effort in this direction, we have been working on the analysis and extraction of detailed test case execution data and source code history, via extending TravisTorrent [78], a tool for analysis of build logs of systems using Travis CI.

Further, as we discussed, the pairwise ranking of test cases using ACER algorithm provides better results compared to state-of-the-art ranking libraries in the context of test case prioritization. It would be interesting to perform extended evaluation between the two methods to check whether or not the similar results can be achievable in a more general context, i.e., pairwise ranking fares better than state-of-the-art ranking model in a more general context.

REFERENCES

[1] P. M. Duvall, S. Matyas, and A. Glover, *Continuous Integration: Improving Software Quality and Reducing Risk.* London, U.K.: Pearson Education, 2007.

[2] J. Humble and D. Farley, *Continuous Delivery: Reliable Software Releases Through Build, Test, and Deployment Automation*. London, U.K.: Pearson Education, 2010.

[3] G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold, "Prioritizing test cases for regression testing," *IEEE Trans. on Softw. Eng.*, vol. 27, no. 10, pp. 929–948, Oct. 2001.

[4] J.-M. Kim and A. Porter, "A history-based test prioritization technique for regression testing in resource constrained environments," in *Proc. 24th Int. Conf. Softw. Eng.*, 2002, pp. 119–129.

[5] B. Busjaeger and T. Xie, "Learning for test prioritization: An industrial case study," in *Proc. 24th ACM SIGSOFT Int. Symp. Found. Softw. Eng.*, 2016, pp. 975–980.

[6] A. Bertolino, A. Guerriero, B. Miranda, R. Pietrantuono, and S. Russo, "Learning-to-rank vs ranking-to-learn: Strategies for regression testing in continuous integration," in *Proc. 42nd Int. Conf. Softw. Eng.*, 2020, pp. 1–12.

[7] C. Zhang, Y. Zhang, X. Shi, G. Almpanidis, G. Fan, and X. Shen, "On incremental learning for gradient boosting decision trees," *Neural Process. Lett.*, vol. 50, no. 1, pp. 957–987, 2019.

[8] D. Silver et al.,, "A general reinforcement learning algorithm that masters chess, shogi, and go through self-play," *Science*, vol. 362, no. 6419, pp. 1140–1144, 2018.

[9] H. Cai et al., "Real-time bidding by reinforcement learning in display advertising," in *Proc. Tenth ACM Int. Conf. Web Search Data Mining*, 2017, pp. 661–670.

[10] X. Zhao, L. Zhang, Z. Ding, L. Xia, J. Tang, and D. Yin, "Recommendations with negative feedback via pairwise deep reinforcement learning," in *Proc. 24th ACM SIGKDD Int. Conf. Knowl. Discov. Data Mining*, 2018, pp. 1040–1048.

[11] H. Li, "Learning to rank for information retrieval and natural language processing," *Synth. Lectures Hum. Lang. Technol.*, vol. 4, no. 1, pp. 1–113, 2011.

[12] A. Hill et al., "Stable baselines," 2018. [Online]. Available: https://github.com/hill-a/stable-baselines

[13] P. Dhariwal et al., "Openai baselines," 2017. [Online]. Available: https://github.com/openai/baselines

[14] M. Hoffman et al., "Acme: A research framework for distributed reinforcement learning," 2020, arXiv: 2006.00979. [Online]. Available: https://arxiv.org/abs/2006.00979

[15] H. Spieker, A. Gotlieb, D. Marijan, and M. Mossige, "Reinforcement learning for automatic test case prioritization and selection in continuous integration," in *Proc. 26th ACM SIGSOFT Int. Symp. Softw. Testing Anal.*, 2017, pp. 12–22.

[16] Scientific Toolworks, Inc., "Understand^{TM}," Accessed: Oct. 14, 2020. [Online]. Available: https://www.scitools.com/

[17] C. J. Watkins and P. Dayan, "Q-learning," *Mach. Learn.*, vol. 8, no. 3–4, pp. 279–292, 1992.

[18] V. Mnih et al.,, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, no. 7540, pp. 529–533, 2015.

[19] R. J. Williams, "Simple statistical gradient-following algorithms for connectionist reinforcement learning," *Mach. Learn.*, vol. 8, no. 3–4, pp. 229–256, 1992.

[20] O. Nachum, M. Norouzi, K. Xu, and D. Schuurmans, "Bridging the gap between value and policy based reinforcement learning," in *Proc. Int. Conf. Neural Inf. Process. Syst.*, 2017, pp. 2772–2782.

[21] V. R. Konda and J. N. Tsitsiklis, "Actor-critic algorithms," in *Proc. Int. Conf. Neural Inf. Process. Syst.*, 2000, pp. 1008–1014.

[22] D. Precup, "Eligibility traces for off-policy policy evaluation," in *Proc. Seventeenth Int. Conf. Mach. Learn.*, 2000, p. 80.

[23] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*. Cambridge, MA, USA: MIT Press, 2018.

[24] V. Mnih et al., "Playing atari with deep reinforcement learning," 2013, arXiv:1312.5602.

[25] T. P. Lillicrap et al., "Continuous control with deep reinforcement learning," 2015, arXiv:1509.02971.

[26] V. Mnih et al., "Asynchronous methods for deep reinforcement learning," in *Int. Conf. Mach. Learn.*, 2016, pp. 1928–1937.

[27] Z. Wang et al., "Sample efficient actor-critic with experience replay," 2016, arXiv:1611.01224.

[28] Y. Wu, E. Mansimov, R. B. Grosse, S. Liao, and J. Ba, "Scalable trust-region method for deep reinforcement learning using kronecker-factored approximation," in *Proc. Int. Conf. Neural Inf. Process. Syst.*, 2017, pp. 5279–5288.

[29] S. Fujimoto, H. Van Hoof, and D. Meger, "Addressing function approximation error in actor-critic methods," 2018, arXiv: 1802.09477.

[30] T. Haarnoja, A. Zhou, P. Abbeel, and S. Levine, "Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor," 2018, arXiv: 1801.01290.

[31] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," 2017, arXiv: 1707.06347.

[32] J. Schulman, S. Levine, P. Abbeel, M. Jordan, and P. Moritz, "Trust region policy optimization," in *Proc. Int. Conf. Mach. Learn.*, 2015, pp. 1889–1897.

[33] S. Yoo and M. Harman, "Regression testing minimization, selection and prioritization: A survey," *Softw. Testing, Verification Rel.*, vol. 22, no. 2, pp. 67–120, 2012.

[34] D. Suleiman, M. Alian, and A. Hudaib, "A survey on prioritization regression testing test case," in *Proc. 8th Int. Conf. Inf. Technol.*, 2017, pp. 854–862.

[35] M. Khatibsyarbini, M. A. Isa, D. N. Jawawi, and R. Tumeng, "Test case prioritization approaches in regression testing: A systematic literature review," *Inf. Softw. Technol.*, vol. 93, pp. 74–93, 2018.

[36] G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold, "Test case prioritization: An empirical study," in *Proc. IEEE Int. Conf. on Softw. Maintenance*, 1999, pp. 179–188.

[37] L. Tahat, B. Korel, M. Harman, and H. Ural, "Regression test suite prioritization using system models," *Softw. Testing, Verification Rel.*, vol. 22, no. 7, pp. 481–506, 2012.

[38] B. Korel, L. H. Tahat, and M. Harman, "Test prioritization using system models," in *Proc. 21st IEEE Int. Conf. Softw. Maintenance*, 2005, pp. 559–568.

[39] Y. Fazlalizadeh, A. Khalilian, M. A. Azgomi, and S. Parsa, "Incorporating historical test case performance data and resource constraints into test case prioritization," in *Proc. Int. Conf. Tests Proofs*, 2009, pp. 43–57.

[40] H. Park, H. Ryu, and J. Baik, "Historical value-based approach for cost-cognizant test case prioritization to improve the effectiveness of regression testing," in *Proc. Second Int. Conf. Secure Syst. Integration Rel. Improvement*, 2008, pp. 39–46.

[41] H. Srikanth, L. Williams, and J. Osborne, "System test case prioritization of new and regression test cases," in *Proc. Int. Symp. Empirical Softw. Eng.*, 2005, p. 10, doi: 10.1109/ISESE.2005.1541815.

[42] H. Srikanth and L. Williams, "On the economics of requirements-based test case prioritization," *ACM SIGSOFT Softw. Eng. Notes*, vol. 30, no. 4, pp. 1–3, 2005.

[43] J. A. Jones and M. J. Harrold, "Test-suite reduction and prioritization for modified condition/decision coverage," *IEEE Trans. Softw. Eng.*, vol. 29, no. 3, pp. 195–209, Mar. 2003.

[44] D. Jeffrey and N. Gupta, "Test case prioritization using relevant slices," in *Proc. 30th Annu. Int. Comput. Softw. Appl. Conf.*, 2006, pp. 411–420.

[45] L. Sui, J. Dietrich, A. Tahir, and G. Fourtounis, "On the recall of static call graph construction in practice," in *Proc. IEEE/ACM 42nd Int. Conf. Softw. Eng.*, 2020, pp. 1049–1060.

[46] S. Elbaum, G. Rothermel, and J. Penix, "Techniques for improving regression testing in continuous integration development environments," in *Proc. 22nd ACM SIGSOFT Int. Symp. Found. Softw. Eng.*, 2014, pp. 235–245.

[47] J. A. P. Lima, W. D. Mendonça, S. R. Vergilio, and W. K. Assunção, "Learning-based prioritization of test cases in continuous integration of highly-configurable software," in *Proc. 24th ACM Conf. Syst. Softw. Product Line: Vol.*, 2020, pp. 1–11.

[48] J. A. do Prado Lima and S. R. Vergilio, "A multi-armed bandit approach for test case prioritization in continuous integration environments," *IEEE Trans. Softw. Eng.*, early access, 2020, doi: 10.1109/TSE.2020.2992428.

[49] A. Memon et al., "Taming google-scale continuous testing," in *Proc. IEEE/ACM 39th Int. Conf. Softw. Eng.: Softw. Eng. Pract. Track*, 2017, pp. 233–242.

[50] B. Korel and G. Koutsogiannakis, "Experimental comparison of code-based and model-based test prioritization," in *Proc. Int. Conf. Softw. Testing, Verification, Validation Workshops*, 2009, pp. 77–84.

[51] T. B. Noor and H. Hemmati, "A similarity-based approach for test case prioritization using historical failure data," in *Proc. IEEE 26th Int. Symp. Softw. Rel. Eng.*, 2015, pp. 58–68.

[52] D. Marijan, A. Gotlieb, and S. Sen, "Test case prioritization for continuous regression testing: An industrial case study," in *Proc. IEEE Int. Conf. Softw. Maintenance*, 2013, pp. 540–543.

[53] V. H. Durelli et al., "Machine learning applied to software testing: A systematic mapping study," *IEEE Trans. Rel.*, vol. 68, no. 3, pp. 1189–1212, Sep. 2019.

[54] R. Carlson, H. Do, and A. Denton, "A clustering approach to improving test case prioritization: An industrial case study," in *Proc. IEEE Int. Conf. Softw. Maintenance*, 2011, pp. 382–391.

[55] A. R. Lenz, A. Pozo, and S. R. Vergilio, "Linking software testing results with a machine learning approach," *Eng. Appl. Artif. Intell.*, vol. 26, no. 5–6, pp. 1631–1640, 2013.

[56] P. Tonella, P. Avesani, and A. Susi, "Using the case-based ranking methodology for test case prioritization," in *Proc. 22nd IEEE Int. Conf. Softw. Maintenance*, 2006, pp. 123–133.

[57] R. Lachmann, S. Schulze, M. Nieke, C. Seidl, and I. Schaefer, "System-level test case prioritization using machine learning," in *Proc. 15th IEEE Int. Conf. Mach. Learn. Appl.*, 2016, pp. 361–368.

[58] C. J. Burges, "From rankNet to lambdaRank to lambdaMART: An overview," Tech. Rep. MSR-TR-2010–82, 2010.

[59] G. Dulac-Arnold, D. Mankowitz, and T. Hester, "Challenges of real-world reinforcement learning," 2019, *arXiv: 1904.12901*.

[60] G. Dulac-Arnold *et al.*, "Deep reinforcement learning in large discrete action spaces," 2015, *arXiv:1512.07679*.

[61] D. L. Donoho *et al.*, "High-dimensional data analysis: The curses and blessings of dimensionality," *AMS Math Challenges Lecture*, vol. 1, no. 2000, p. 32, 2000.

[62] G. Weisz, P. Budzianowski, P.-H. Su, and M. Gašić, "Sample efficient deep reinforcement learning for dialogue systems with large action spaces," *IEEE/ACM Trans. Audio, Speech, Lang. Process.*, vol. 26, no. 11, pp. 2083–2097, Nov. 2018.

[63] M. Riedmiller *et al.*, "Learning by playing-solving sparse reward tasks from scratch," 2018, *arXiv: 1802.10567*.

[64] D. E. Knuth, *The Art Of Computer Programming: Volume 3: Sorting And Searching*. Redwood City, CA, USA: Addison-Wesley, 1998.

[65] "Gym library," 2020. [Online]. Available: https://github.com/openai/gym

[66] S. Varrette, P. Bouvry, H. Cartiaux, and F. Georgatos, "Management of an academic HPC cluster: The UL experience," in *Proc. Int. Conf. High Perform. Comput. Simul.*, 2014, pp. 959–967.

[67] B. L. Welch, "The generalization of 'student's' problem when several different population variances are involved," *Biometrika*, vol. 34, no. 1/2, pp. 28–35, 1947.

[68] P. A. Games and J. F. Howell, "Pairwise multiple comparison procedures with unequal n's and/or variances: A Monte Carlo study," *J. Educ. Behav. Statist.*, vol. 1, no. 2, pp. 113–125, 1976.

[69] H.-Y. Kim, "Analysis of variance (ANOVA) comparing means of more than two groups," *Restorative Dent. Endodontics*, vol. 39, no. 1, pp. 74–77, 2014.

[70] K. O. McGraw and S. P. Wong, "A common language effect size statistic," *Psychol. Bull.*, vol. 111, no. 2, pp. 361–365, 1992.

[71] A. Arcuri and L. Briand, "A hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering," *Softw. Testing, Verification Rel.*, vol. 24, no. 3, pp. 219–250, 2014.

[72] C. Kang *et al.*, "Learning to re-rank web search results with multiple pairwise features," in *Proc. fourth ACM Int. Conf. Web Search Data Mining*, 2011, pp. 735–744.

[73] "Ranklib," 2020. [Online]. Available: https://sourceforge.net/p/lemur/wiki/RankLib/

[74] H. Do and G. Rothermel, "On the use of mutation faults in empirical assessments of test case prioritization techniques," *IEEE Trans. Softw. Eng.*, vol. 32, no. 9, pp. 733–752, Sep. 2006.

[75] Q. Luo, K. Moran, D. Poshyvanyk, and M. Di Penta, "Assessing test case prioritization on real faults and mutants," in *Proc. IEEE Int. Conf. Softw. Maintenance Evol.*, 2018, pp. 240–251.

[76] M. Beller, G. Gousios, and A. Zaidman, "Oops, my tests broke the build: An explorative analysis of travis CI with GitHub," in *Proc. IEEE/ACM 14th Int. Conf. Mining Softw. Repositories*, 2017, pp. 356–367.

[77] T. Akiba, S. Sano, T. Yanase, T. Ohta, and M. Koyama, "Optuna: A next-generation hyperparameter optimization framework," in *Proc. 25th ACM SIGKDD Int. Conf. Knowl. Discov. Data Mining*, 2019, pp. 2623–2631.

[78] M. Beller, G. Gousios, and A. Zaidman, "Travistorrent: Synthesizing travis CI and github for full-stack research on continuous integration," in *Proc. 14th Int. Conf. Mining Softw. Repositories*, 2017, pp. 447–450.

Mojtaba Bagherzadeh received the PhD degree in computer science from Queen's University, Canada, in 2019. He is currently a NSERC postdoctoral fellow with the School of Electrical Engineering and Computer Science, University of Ottawa. He has several years of experience as a software developer with IBM Canada and Advance Mobile Payment (AMP). He was working on HPC cluster management solutions with IBM and started developing a performance analyzer tool for huge HPC clusters as an initiative. Before joining IBM, he was a team lead with AMP where he led the development of mobile payment and transaction processing solutions. His research interests include empirical software engineering, model-driven engineering, and the debugging and testing of cyber-physical and machine learning-based systems.

Nafiseh Kahani received the bachelor's degree in software engineering, the master's degree in information technology, and the PhD degree in computer science from Queen's University. She is currently an assistant professor with the Department of Systems and Computer Engineering, Carleton University. Her research interests include model-driven development, applications of machine learning in software engineering, software testing, and software security.

Lionel Briand (Fellow, IEEE) is currently a professor of software engineering and has shared appointments betweenthe School of Electrical Engineering and Computer Science, University of Ottawa, Canada, and The SnT centre for Security, Reliability, and Trust, University of Luxembourg. His research interests include model-driven development, testing and verification, search-based software engineering, requirements engineering, and empirical software engineering. He is the head of the SVV Department, SnT Centre, and a Canada research chair in Intelligent Software Dependability and Compliance (Tier 1). He holds an ERC Advanced Grant, the most prestigious European individual research award, and has conducted applied research in collaboration with industry for more than 25 years, including projects in the automotive, aerospace, manufacturing, financial, and energy domains. He is a fellow of the IEEE and ACM. He was the recipient of the IEEE Computer Society Harlan Mills Award in 2012 and the IEEE Reliability Society Engineer-of-the-Year Award in 2013 for his work on model-based verification and testing.

▷ **For more information on this or any other computing topic, please visit our Digital Library at** www.computer.org/csdl.