Evaluating Machine Learning-Based Test Case Prioritization in the Real World: An Experiment with SAP HANA

Jeongki Son
SAP Labs Korea
Seoul, South Korea
jeongki.son@sap.com

Gabin An
KAIST
Daejeon, South Korea
gabin.an@kaist.ac.kr

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

Shin Yoo
KAIST
Daejeon, South Korea
shin.yoo@kaist.ac.kr

***Abstract*—Test Case Prioritization (TCP) aims to find orderings of regression test suite execution so that failures can be detected as early as possible. Recently, Machine Learning (ML) based techniques have been proposed and evaluated using open-source projects and their test histories. We report our evaluation of these ML-based TCP techniques, both Reinforcement Learning (RL) and Supervised Learning (SL) based ones, using the industrial testing data collected from SAP HANA, a large-scale database management system. Specifically, our study compares 37 different TCP techniques, including 14 RL models on two datasets, 4 SL models, and 5 non-ML baselines, using real-world testing data of SAP HANA collected over eight months. Our evaluation focuses on both the performance and cost-efficiency of these techniques in the context of Continuous Integration for large-scale industrial projects. The results reveal that while RL models show promising performance, they require significant training time. RL models with sampled data offer a balance between performance and efficiency. Interestingly, the best-performing RL model outperformed or matched non-ML baselines. However, the gradient-boosted SL technique consistently outperformed both RL models and baselines in terms of effectiveness and efficiency, even with complete retraining at each test cycle. Despite RL's capability for incremental learning, it demands substantial training time and still falls short in accuracy compared to SL. Our findings suggest that, even in a large-scale industrial setting, fully retraining an SL model for each cycle proves to be the most effective and efficient approach for TCP, offering superior performance and cost-efficiency compared to RL and traditional methods.**

***Index Terms*—test case prioritization, machine learning**

I. Introduction

In the software development lifecycle, regression testing plays a crucial role in ensuring that new code changes introduced to a program do not break existing functionalities [1]. As software systems grow and evolve, their regression test suites also tend to grow larger, leading to inefficiencies and extended testing times. The increasing size of the regression test suite can have a significant impact on the efficiency of the overall CI/CD process, as testing is the most time-consuming task [2]. This issue is especially amplified for a large-scale industrial database management system, such as SAP HANA (with >36M LoC) [3], which are essential for the functioning of other systems and thus require thorough quality assurance measures, including extensive regression testing. To reduce the high cost of testing and also to provide test results back to the developers more quickly, SAP HANA employs a multi-layered testing strategy, including local, pre-submit, post-submit, and extended testing phases [3]. Further, the existing test infrastructure for SAP HANA leverages a container-based model that enables parallel execution of tests across both on-premises and cloud environments [4]. However, despite the parallelization, the pre-submit testing phase still remains a bottleneck in the development workflow, often taking several hours due to the heavy load of SQL query testing as well as the frequency at which new changes are submitted to be tested.

In this scenario, Test Case Prioritization (TCP) can play an important role by ordering the test cases to detect faults as early as possible [1], [5], in turn enabling quicker reporting of test failures to developers. Additionally, when testing resources are limited, TCP can also be used to select test cases by running the top-k ranked test cases [6]. Due to its usefulness, numerous studies have integrated TCP into industrial scenarios [7]–[12]: they typically prioritize test cases based on their coverage (so that higher coverage is achieved as early as possible) or historical data (so that a test that has not been executed for the longest time is given higher priority). Recently, beyond simple dynamic or historical analysis, Machine Learning (ML) techniques have been actively proposed and applied to TCP [13], as the learning-to-rank models fit well with the TCP problem. However, while various ML-based TCP methods are being proposed and tested on open-source projects [14], there remains a gap in comprehensive comparative studies of these ML-based TCP techniques on a large-scale industrial project.

In this study, we share our experience of employing a range of ML-based TCP techniques using historical test case execution data from SAP HANA. Our analysis includes the empirical evaluation of seven reinforcement learning algorithms following the formulation of Bagherzadeh et al. [15] and tree-based supervised learning algorithms including three state-of-the-art gradient boosting algorithms, namely CatBoost [16], XGBoost [17], and LightGBM [18], along with Random Forest [19]. The results show that state-of-the-art gradient-boost supervised learning techniques can outperform RL-based TCP with the pairwise ranking model, which is the best-performing

979-8-3315-0814-2/25 © 2025 IEEE

ICST 2025, Naples, Italy
Industry Track

ranking model for RL.

II. Background and Related Work

This section introduces the background on test case prioritization and reviews some of the closest related work on the use of machine learning techniques to solve this problem.

A. Test Case Prioritization (TCP)

There are broadly two distinct approaches to TCP: *general* and *version-specific* TCP [20]. General TCP techniques involve ordering test cases within a test suite T for a program P, to find an order that remains effective across future versions of P. These techniques typically depend on a broad range of program or test information, such as test coverage [20], but are known to be less effective in a modern Continuous Integration (CI) process where incremental changes are made to software frequently. Version-specific TCP techniques, on the other hand, focus on ordering test cases specifically to test changes from a specific version P to another version P'. These techniques are therefore mostly change-aware [21], and closely consider the specific changes or modifications introduced in the new version of the program, P'. In this work, we also focus on the change-aware TCP to facilitate the regression testing process within the CI process of SAP HANA. For more comprehensive information about TCP, please refer to the recent survey [5].

B. Machine Learning based TCP

Machine Learning (ML) techniques have been widely adopted to solve the TCP problem, as surveyed in Pan et al. [13]. Most ML-based TCP techniques utilize the *Learning-To-Rank* [22] approach, where a model takes a set of test cases, characterized by various features, and learns to predict the best order to execute them to detect faults as early as possible, from historical failure data.

Reinforcement Learning (RL) and Supervised Learning (SL) are the major ML techniques used in TCP [14]. Bagherzadeh et al. [15] have applied pointwise, pairwise, and listwise learning-to-rank formulations to model TCP as an RL problem, and evaluated 10 state-of-the-art RL algorithms, resulting in a set of 21 different RL configurations. They demonstrated that their best RL models outperformed the accuracy of the ranking model based on Multiple Additive Regression Trees (MART), a.k.a. Gradient Boosting Trees, which exhibited superior performance compared to the RL models in Bertolino et al. [23]. Yaraghi et al. [24] compare multiple supervised-learning-based ranking models, such as MART and Random Forest, using a comprehensive set of features and various publicly available CI datasets. The study found that Random Forest with full features shows significantly better performance than other ranking models, including MART. Zhao et al. [14] conducted a comparative study on the performance of 11 different ML-based TCP (both SL and RL) using 11 GitHub open-source projects and showed that the pre-trained MART model performs the best, suggesting that cross-project training helps improve the TCP effectiveness.

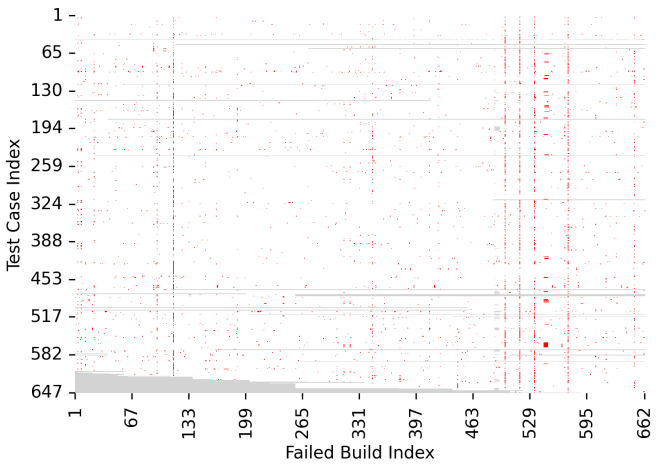

Fig. 1. Visualization of test case results across 662 failed testing cycles used in the experiment: red dots indicate failed tests, white represents passed tests, and gray marks tests that were not executed

All of the aforementioned studies [14], [15], [24] utilized the implementation of the SL-based ranking models available in the RankLib[1] library rather than using the latest gradient boosting models, such as XGBoost [17] or CatBoost [16], despite XGBoost known to outperform other SL-based ranking models for TCP [25]. To the best of our knowledge, we present the first comparison between the state-of-the-art RL-based TCP techniques [15] and the state-of-the-art gradient boosting tree models using large-scale industry data.

III. Experiment Design

In this study, we utilize the industry CI data from SAP HANA to perform a comprehensive comparison between the state-of-the-art RL algorithms and ensemble-based learning algorithms, including Boosting (Gradient Boosting Trees) and Bagging (Random Forest). We assess the effectiveness of these algorithms using the readily available and lightweight features extracted from the historical CI data in SAP HANA. This section offers detailed information about the dataset and the features used for evaluation and explains the specific implementation details of the ML algorithms used.

A. Data Collection and Preparation

To assess the performance of the ML-based TCP algorithms, we gathered data from the failed testing cycles of the pre-submit testing results in SAP HANA. The data was collected during the period starting from April 25, 2023, to December 22, 2023. For each testing cycle, we recorded the list of executed test cases and their verdicts, i.e., whether they passed or failed, along with the features collected for each test case. The plot in Figure 1 visualizes failed test cases across failed testing cycles. On average, there are about 5.68 failures per build, with a maximum of 291 failures in a single build (note that we only consider failed cycles here). The plot indicates diverse

[1]https://github.com/codelibs/ranklib

failures across test cases, with 85% of test cases failing at least once. This suggests that the TCP problem in our context can be considered non-trivial as there is a broad range of issues rather than a concentration on specific issues. Using the collected 662 failed cycles with 647 distinct test cases, we evaluate RL-based TCP techniques using the *incremental learning* approach: as additional failed cycles are gathered, the training data grows, and the model is incrementally updated. Following Bagherzadeh et al. [15], we first choose all failing tests accumulated up to the current cycle, and randomly sample an equal number of passing tests, to balance the dataset as well as to improve efficiency. For the SL-based techniques, we use the whole dataset accumulated up to each test cycle for training, because they are fast enough.

B. Features

A recent study on TCP conducted by Yaraghi et al. [24] reports that the features that rely solely on test execution history are the easiest to collect. Furthermore, they also found that the simple test history features have more impact on the effectiveness of TCP compared to the more expensive coverage features. Since the codebase of SAP HANA is extensive, we focus primarily on using the most lightweight and readily available features, which are the history-based features, as recommended by Yaraghi et al. [24].

In addition, we do consider test coverage as it can provide valuable insights for version-specific TCP. We use the latest dynamic coverage information of each test case, as the periodically updated coverage data [26] is available for SAP HANA. Finally, we include lexical similarities between the changed files and test case names to capture the relevance between the test cases and the code change.

Overall, we use a total of 25 features to represent each test case for a given testing cycle. The used features are broadly divided into the following two categories: **Test-Related** and **Change-Related**. The definitions of all used features are as follows:

1) Test-Related Features: While these features have been taken from existing work on TCP [7], [24], [25], [27], the definitions of some features have been slightly adjusted to fit our context. Features with a time window are collected with $d \in \{10, 30, 90\}$.

- **Age (d days)** [7], [24]: The ID difference between the current testing cycle and the earliest cycle that executed a test case within the last d days.
- **Last Fail Age (d days)** [24], [27]: The ID difference between the current testing cycle and the latest cycle where a test case failed within the last d days.
- **Last Transition Age (d days)** [24], [27]: The ID difference between the current testing cycle and the latest cycle where the verdict of a test case changed from fail to pass or vice versa.
- **Average Execution Time (d days)** [24], [25]: The average of the execution times of a test case in the last d days.
- **Max Execution Time (d days)** [24]: The maximum of the execution times of a test case in the last d days.
- **Fail Rate (d days)** [24]: The ratio of the number of failed executions to the total number of executions of the test case.
- **Transition Rate (d days)** [24], [27]: The rate of transitions of the test case verdicts.
- **Last Verdict** [24]: The result of the last execution of a test case (failing: -1, passing: 1).
- **Last Execution Time** [24]: The duration of the last execution.

2) Change-Related Features: These features concern the changes between previous version P and the current version under test, P'.

- **Lexical Similarity** [27]: The lexical similarity between a set of changed file names (i.e., docs) and a test case name (i.e., query) using TF-IDF and cosine similarity. There are multiple Information-Retrieval-based TCP techniques, such as REPiR [28], that consider the test source code as documents and the change information as queries. However, in SAP HANA, we do not use the test case code but only their names because the test cases in SAP HANA are written in Python and mostly invoke SQL queries, while the software itself is mostly written in C++.
- **Test Coverage of Modified Code** [26]: The proportion of changed lines covered by a test case. Test coverage is measured weekly in SAP HANA, and the latest coverage data for each test case is used. Since this could lead to slight mismatches in line numbers over time, a ± 10 line padding is applied to the changed lines to cater for potential discrepancies.

C. Implementation of ML Models

We investigate both RL- and SL-based TCP. For RL, we explore seven different techniques, based on the ranking models proposed by Bagherzadeh et al. [15]. For SL, our focus is on ensemble-based decision tree models. This section describes the implementation details for each of the ML techniques.

1) Implementation of Reinforcement Learning Algorithms: We adapt RL-based TCP techniques proposed by Bagherzadeh et al. [15] for our dataset. Specifically, we use the RL algorithm implementation from the Stable-Baselines3 (v2.2.1) [29] and configure the RL environment using the Gym library (v0.26.2) [30]. We use seven different state-of-the-art RL algorithms to train the agents: Advantage Actor-Critic (A2C) [31], Deep Deterministic Policy Gradient (DDPG), Deep Q-Networks (DQN) [32], Proximal Policy Optimization (PPO) [33], Soft Actor-Critic (SAC) [34], Twin Delayed DDPG (TD3) [35], and Trust Region Policy Optimization (TRPO) [36].

Pointwise Ranking Model: The RL agent considers a single test case and assigns a score from the continuous range between 0 and 1 to each test as an action. Tests are then sorted according to their scores, and the reward is computed based on the difference between the normalized optimal rank of the test case and their current score. Only RL algorithms that support a continuous action space (i.e., the priority score), can be used

for the pointwise ranking: A2C, DDPG, PPO, SAC, TD3, and TRPO.

Pairwise Ranking Model: The RL agent considers a pair of tests and performs a step-wise bubble sort based on the comparison of priority between the pair as an action. The reward is computed based on the correctness of priority comparison with respect to the optimal order. Since the action space is discrete (i.e., to determine which of the two test cases has higher priority), RL algorithms that support a discrete action space can be used for pairwise ranking: A2C, DQN, PPO, and TRPO.

Listwise Ranking Model: The RL agent considers the entire test suite, and gradually construct the ordering by selecting the index of the test case with the highest priority as an action. The reward is computed based on the difference between the normalized optimal rank of the test case and the normalized rank in the current ordering. Since the action is discrete (i.e., integer index), only RL algorithms that support a discrete action space can be applied for the listwise ranking: A2C, DQN, PPO, and TRPO.

The number of steps (the total number of samples to train on) was set based on the worst case as in the previous work [15] with some modifications. For instance, in the pairwise ranking, we employed a bubble sort algorithm with quadratic complexity, which requires n^2 steps per learning episode, where n is the number of test cases in each run. Also, we fixed the number of episodes as 5 for all experiments. All algorithms were trained using their default parameters except for Listwise with DQN, where we set the buffer size to 10,000 due to memory constraints.

2) Implementation of Decision Tree Based Ensemble Models: Decision-tree-based ensemble models are an effective and robust approach for a variety of regression and classification tasks. These SL models are also commonly employed for ranking tasks, due to their adaptability to distinct objective functions. We utilize state-of-the-art gradient boosting decision tree algorithms (libraries), XGBoost [17], CatBoost [16], and LightGBM [37], along with a decision-tree-based bagging algorithm, Random Forest, using the implementation provided by the `scikit-learn` [38] library.

We employ the ranker library for XGBoost, CatBoost, and LightGBM, and the classifier for RandomForest, with their default parameters. In XGBoost (v2.0.3), XGBRanker employs the 'rank:ndcg' objective function by default. This function is based on LambdaMART [39], a pairwise learn-to-rank algorithm, and utilizes a surrogate gradient derived from the Normalized Discounted Cumulative Gain (NDCG) metric. The default objective function for CatBoostRanker in CatBoost (v1.2.2) is YetiRank, which also corresponds to pairwise ranking. Similarly, the LGBMRanker in LightGBM (v4.3.0) uses LambdaRank [40] as its default objective, which also aligns with pairwise ranking. The RandomForestClassifier model in `scikit-learn` (v1.2.2) uses Gini impurity as its default criterion for node splitting. While primarily designed for classification tasks, it can be adapted for ranking in a pointwise approach by utilizing the probability values assigned to each test case.

D. Data Sampling

As shown in prior work [15], a significant drawback of RL (particularly the pairwise ranking model) is the high execution cost associated with training. To mitigate this, we also evaluate the efficacy of the RL model using a sampled dataset to reduce costs. For sampling, we employ a simple random sampling method, aiming to balance the number of passed and failed test cases. Specifically, we randomly select S test cases from both the set of passed test cases (S_p) and the set of failed test cases (S_f), where the number of samples, S, is determined by the smaller of the two sets: $S = min(|S_p|, |S_f|)$. This approach helps us maintain an equal representation of both outcomes in our training data, which we believe may contribute to more efficient model training.

E. Baseline Approaches

We set our baselines by sorting test cases based on the four most important features from the SL model: Lexical Similarity, Test Coverage of Modified Code, Max Execution Time (90 days), and Max Execution Time (30 days). We sort test cases in descending order based on these features, assuming higher values might indicate higher failure probability (see Section III-B). In addition, we also implement a *Comprehensive* approach, which starts sorting with the most important feature, and breaks ties with the next important features until tests are not tied or no features are left to break ties. These *non-ML* baselines only rely on sorting, though guided by the SL model for feature selection, relies on simple sorting for test case prioritization, making it easily interpretable.

F. Metrics for Evaluation

To assess the TCP performance of each configuration, we employ the Average Percentage of Faults Detected (APFD) [20] metric, which is the most widely used evaluation metric for TCP [13]. In addition, we use two general ranking evaluation metrics, Mean Average Precision (MAP) and Mean Reciprocal Rank (MRR), as supplementary metrics.

- Average Percentage of Faults Detected (APFD): This metric measures the effectiveness of a test case ordering in detecting faults at the earliest. It is calculated using the formula [13]:

$$APFD = 1 - \frac{\sum_{i=1}^{m} TF_i}{nm} + \frac{1}{2n}$$

  where m is the total number of faults (test failures) that can be detected by the test suite, n is the total number of test cases, and TF_i is the rank of the first test case that reveals the ith fault. A higher APFD value indicates that faults are detected earlier, which is desirable. When reporting the APFD, we average the APFD scores across all failing testing cycles in our test data.
- Mean Average Precision (MAP): MAP is the mean of the Average Precision (AP) of the failing test cases. The AP for a single cycle is defined as the average of the

precisions at the points in the ranking where a failing test is found.

- Mean Reciprocal Rank (MRR): MRR is the mean of the reciprocal ranks of the first failing test in the context of TCP. Unlike APFD and MAP, MRR is more focused on the rank of the first fault detected, rather than the overall fault detection rate.
- Training time (min.): For RL, we recorded the training time for each run during training, as it can be updated in real-time. For SL, which uses a pre-collected, fixed dataset and doesn't update in real-time, we simulated real-time updates by training in batches up to the run we wanted to test. For example, to test run k, we trained on runs 1 to k-1 and measured the total training time. This approach allowed for a fair comparison between the two methods.
- Prediction time (min.): we measured for both RL and SL models by testing on a single run. We recorded the time it took for the model to return an output (ranking) given a set of test cases from the specific run.

IV. Results

This section presents the results of our experiments, which were conducted on an Apple M2 Pro chip with 8 cores, and 16GB memory.

A. Performance of RL algorithms

Table I summarizes the average performance and cost across three key metrics (APFD, MAP, and MRR) for all studied configurations. For brevity, we hereafter use [Ranking Model Abbreviation]-[RL Algorithm] format to denote a configuration. For example, "PA-A2C" means that the configuration uses the pairwise ranking model and the A2C as the RL algorithm. Models trained on sampled datasets are marked with a †.

1) RL-based Configurations: Among RL-based TCP techniques, pairwise ranking models consistently achieve the highest APFD as shown in Table I. PA-A2C shows the best performance among the RL techniques, followed closely by PA-DQN. The dominance of pairwise ranking models, which is in line with the results reported by Bagherzadeh et al. [15], likely stems from the smaller observation space for RL, as well as the step-wise sorting action that explicitly swaps positions of the tests. On the other hand, pointwise and listwise ranking models tend to perform worse than the pairwise ranking model. To highlight the performance difference, we have computed the Common Language Effect Size (CLES) [41] between the best (PA-A2C) and the worst (PO-PPO) configurations, which is 0.912. It means that a randomly chosen ordering from the best group has a 91.2% probability of having a higher APFD than one randomly chosen from the worst group.

Regarding the training time, several configurations (LI-DQN, PO-TRPO, and PO-PPO) appear to be efficient, with LI-DQN being the most efficient. PA-A2C had the longest training time, with some of the pairwise configurations with discrete action space algorithms (DQN, PPO, and TRPO) also being slow due to the quadratic complexity of the underlying bubble sort algorithm. Based on these results, we found that the LI-DQN and PO-TRPO configurations were efficient in terms of training time for future online learning in the CI environment. However, these configurations showed lower APFD performance. Additionally, we identified the potential for improving the pairwise configuration by using a sorting algorithm with a time complexity of $nlogn$, such as merge sort. Based on these findings, we cannot definitively conclude which configuration is superior.

In terms of prediction time, configurations with pointwise ranking models are the most efficient. Welch's ANOVA [42], confirmed by Games-Howell post-hoc test [43] with the significance level of $\alpha = 0.05$[2], suggests that PO-A2C, PO-PPO, and PO-TRPO are the most efficient for prediction, requiring negligible times (worst case: 0.28 seconds). In contrast, LI-PPO has the worst prediction time, which is likely due to the model repeatedly selecting dummy test cases inserted to keep the size of the observation space consistent across multiple test cycles under incremental learning, combined with the larger observation space of listwise ranking models.

2) Ranking Models: As discussed earlier, the pairwise model outperforms the pointwise and listwise models in terms of ranking accuracy (APFD). We have further analyzed whether this trend applies across different RL algorithms. For each of the three RL algorithms that can be applied to all three ranking models (i.e., A2C, PPO, and TRPO), we have performed Welch's ANOVA between ranking models, followed by the Games-Howell post-hoc test if the means are not equal. Below, $>$ indicates statistically significant differences, whereas $=$ denotes differences that are not statistically significant ($p > 0.05$).

- A2C: PA $>$ LI = PO
- PPO: LI = PA $>$ PO
- TRPO: PA = LI $>$ PO

The tests reveal statistically significant differences in three RL algorithms, with pairwise models showing better performance than pointwise and listwise models. Also, regardless of the specific RL algorithm (A2C, PPO, and TRPO), pairwise models consistently outperform pointwise and listwise approaches. Listwise approaches likely suffer from a higher dimensional observation space, requiring more training data compared to pairwise and pointwise methods, as also pointed out by existing work [15]. Additionally, pairwise approaches excel at capturing relative relationships between test cases by considering them in pairs and performing swaps, whereas pointwise approaches focus on one individual test at a time.

3) RL Algorithms: To assess which RL algorithm performs best in terms of APFD across, we performed Welch's ANOVA and Games-Howell post-hoc tests again on the three ranking

[2]Given that one-way ANOVA assumes homogeneity of variances (homoscedasticity) across groups, we assess this assumption using Bartlett's test [44]. A statistically significant result ($p < 0.001$) indicates heterogeneity of variance, supporting the alternative hypothesis of Bartlett's test: at least one group variance is different. Therefore, we employ Welch's ANOVA, which does not require the assumption of equal variances. When Welch's ANOVA result is significant ($p < 0.05$), we conduct the Games-Howell post-hoc test, which is appropriate when the assumption of equal variances is violated, to determine whether the difference in performance is statistically significant.

TABLE I
ACCURACY, TRAINING TIME, PREDICTION TIME (AVERAGE)

Config.	APFD	MAP	MRR	Training time (min.)	Prediction time (min.)
PO-A2C	0.441	0.017	0.031	0.146	0.001
PO-DDPG	0.466	0.017	0.020	1.565	0.001
PO-PPO	0.310	0.012	0.011	0.138	0.001
PO-SAC	0.390	0.022	0.037	2.401	0.002
PO-TD3	0.451	0.024	0.044	1.550	0.002
PO-TRPO	0.357	0.013	0.015	0.092	0.001
PA-A2C	0.750	0.058	0.080	8.706	0.586
PA-DQN	0.523	0.030	0.047	6.399	0.400
PA-PPO	0.462	0.016	0.021	8.272	0.591
PA-TRPO	0.466	0.017	0.025	5.500	0.611
LI-A2C	0.468	0.018	0.020	0.569	0.089
LI-DQN	0.455	0.017	0.023	0.036	0.324
LI-PPO	0.466	0.017	0.020	0.665	6.961
LI-TRPO	0.465	0.017	0.022	0.615	0.038
PO-A2C†	0.471	0.014	0.017	0.058	0.001
PO-DDPG†	0.467	0.017	0.033	0.635	0.001
PO-PPO†	0.473	0.015	0.018	0.062	0.001
PO-SAC†	0.513	0.018	0.028	1.241	0.001
PO-TD3†	0.475	0.016	0.024	0.644	0.001
PO-TRPO†	0.482	0.015	0.017	0.040	0.001
PA-A2C†	0.541	0.033	0.055	0.065	0.589
PA-DQN†	0.537	0.050	0.080	0.040	0.401
PA-PPO†	0.493	0.030	0.048	0.067	0.589
PA-TRPO†	0.653	0.063	0.098	0.046	0.594
LI-A2C†	0.465	0.015	0.020	0.271	3.844
LI-DQN†	0.480	0.017	0.027	0.217	0.520
LI-PPO†	0.469	0.014	0.017	0.345	4.868
LI-TRPO†	0.480	0.018	0.034	0.294	0.772
CatBoost	0.808	0.090	0.128	1.679	1.368e-05
LightGBM	0.815	0.097	0.133	0.016	1.851e-05
XGBoost	0.801	0.083	0.118	0.018	1.270e-05
RandomForest	0.459	0.014	0.018	0.640	1.041e-04
Lexical Similarity	0.515	0.075	0.109	-	-
Test Coverage of Modified Code	0.705	0.142	0.198	-	-
Max Execution Time (90 days)	0.743	0.044	0.070	-	-
Max Execution Time (30 days)	0.747	0.049	0.080	-	-
Comprehensive	0.580	0.086	0.120	-	-

Note: Models with † were trained on undersampled dataset.

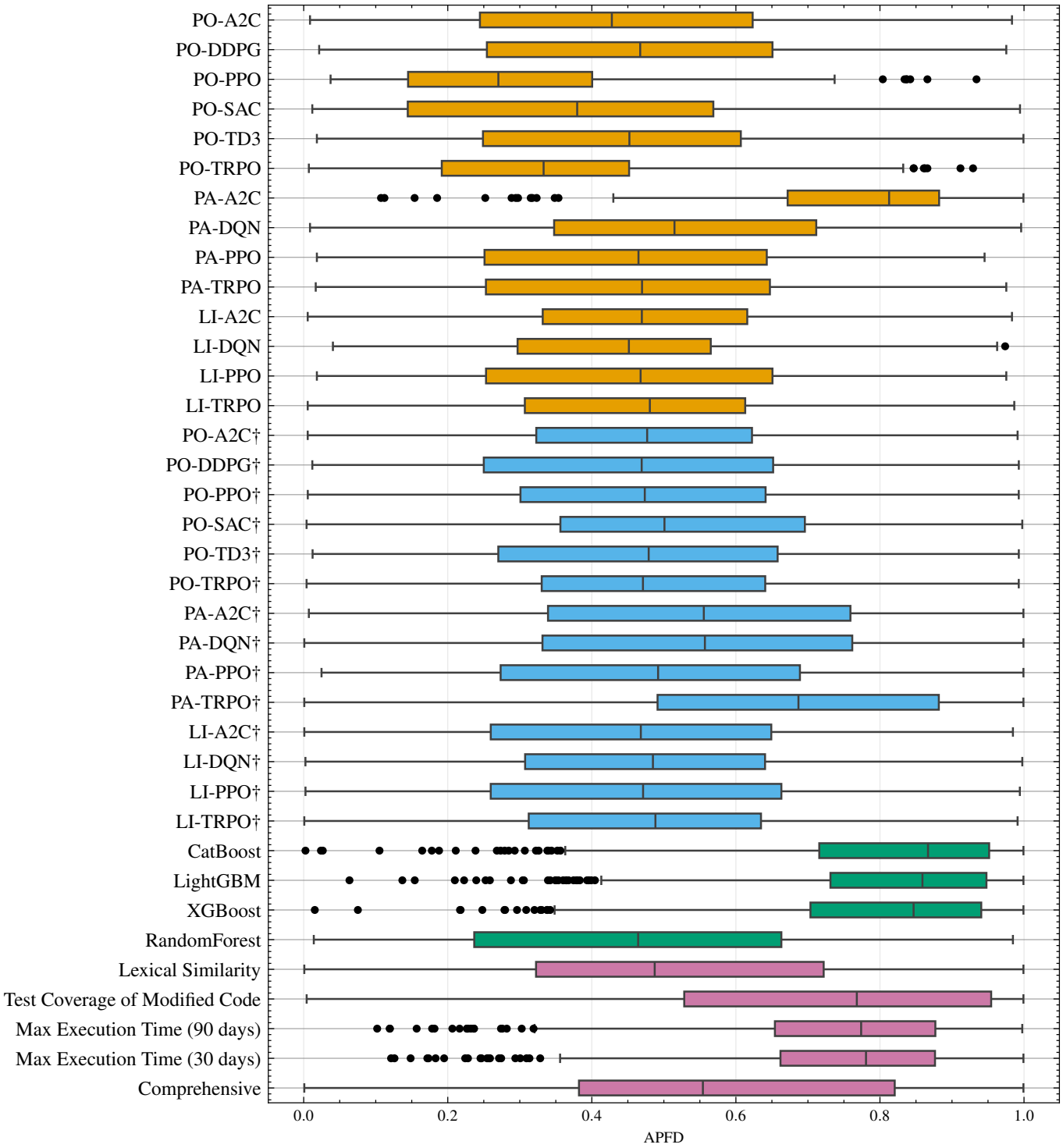

Fig. 2. Comparison of APFD values

models (pointwise, pairwise, and listwise) and compared the RL algorithms. The results indicated no significant differences in performance between RL algorithms for both the pointwise and listwise ranking models. However, for the pairwise ranking model, we observed statistically significant performance variations between RL algorithms:

- PA: A2C > DQN = TRPO = PPO

We performed the same analysis to compare the training time, and observed statistically significant variations across all ranking models.

- PO: SAC > DDPG = TD3 > A2C > PPO > TRPO
- PA: A2C > PPO > DQN > TRPO
- LI: PPO > TRPO > A2C > DQN

Finally, we also performed a similar analysis on prediction time. The results show a significant difference in prediction time across all ranking models.

- PO: SAC > TD3 > DDPG > TRPO = PPO = A2C
- PA: TRPO > A2C = PPO > DQN
- LI: PPO > A2C > TRPO > DQN

Our findings consistently demonstrate the superiority of pairwise models in terms of APFD over pointwise and listwise ranking models. While pairwise configurations may not exhibit the absolute fastest training times, we argue that their accuracy gains outweigh this trade-off. Notably, PA-A2C emerges as the most effective configuration.

4) Data Sampling: As shown in our results and in previous work [15], one major issue with RL compared to SL is that it takes a very long time to train. Therefore, as discussed in Section III-D, we also conducted the same experiments using a balanced dataset. We believed that utilizing a sampled dataset could be both more efficient and effective if it produced similar results to those obtained from the full dataset.

In our experiments with the sampled dataset, similar to the full dataset, the pairwise ranking model consistently achieves the highest APFD, as indicated in Table I. Among the RL models, PA-TRPO† demonstrates the highest ranking accuracy, with PA-A2C† and PA-DQN† following. As seen in the findings of Bagherzadeh et al. [15]. [15] and our initial experiment, the superior performance of the pairwise ranking model is likely due to its approach of considering the ranking between two test cases simultaneously. Conversely, the pointwise and listwise ranking models tend to underperform compared to the pairwise ranking model. To further highlight the performance disparities, we calculated the Common Language Effect Size (CLES) between the best configuration (PA-TRPO†) and the worst configuration (PO-PPO†), which is 0.705. This indicates that PA-TRPO† outperformed in 70% of the test cycles.

In terms of training time, the configurations PA-DQN†, PO-TRPO†, and PA-TRPO† demonstrated notable efficiency, with PA-DQN† being the fastest. Conversely, PO-SAC† required the longest training duration, and some configurations involving continuous action space algorithms (DDPG, TD3, SAC) were also relatively slow. Based on these observations, we suggest using PA-DQN†, PO-TRPO†, and PA-TRPO† (all under 2.75 seconds) for future online learning applications in the CI environment.

Pointwise ranking models are the most efficient in terms of prediction time. Using Welch's ANOVA and the Games-Howell post-hoc test, we found that the PO-A2C†, PO-PPO†, and PO-TRPO† models have the fastest prediction times, taking at most 0.09 seconds. LI-PPO† showed the slowest prediction time, attributable to its frequent selection of dummy test cases and large observation space inherent to listwise ranking approaches.

As mentioned previously, the pairwise model exhibits higher APFD compared to the pointwise and listwise models. To determine if this pattern holds for various RL algorithms, we conducted an additional analysis. Specifically, we applied each of the three RL algorithms (A2C, PPO, and TRPO) to all three ranking models. Subsequently, we performed again Welch's ANOVA to compare the models, followed by the Games-Howell post-hoc test.

- A2C: PA > PO = LI
- PPO: PA = PO = LI
- TRPO: PA > PO = LI

The experiments reveal significant APFD differences between ranking models in A2C and TRPO, with pairwise models outperforming pointwise and listwise models. Across all tested RL algorithms (A2C, PPO, and TRPO), pairwise models consistently deliver superior results compared to their pointwise and listwise counterparts. Listwise approaches likely struggle due to their higher dimensional observation space as we mentioned before. Moreover, pairwise considers relative relationships, while pointwise considers each one at a time, just as we did in our experiment on the whole dataset.

We also performed Welch's ANOVA test to compare the performance of different RL algorithms for each ranking model, pointwise, pairwise, and listwise. The results showed no significant performance differences between the ranking algorithms for the pointwise and listwise ranking models, but the pairwise ranking model revealed significant differences between the algorithms. Specifically, PA-TRPO† stands out as the most effective configuration.

- PA: TRPO > A2C = DQN > PPO

We also performed the same analysis to compare the training time, and found no significant difference between RL algorithms across different ranking models.

Finally, we analyzed prediction times. The results show a significant difference across all ranking models.

- PO: SAC > TD3 > DDPG > TRPO = PPO = A2C
- PA: TRPO > A2C = PPO > DQN
- LI: PPO > A2C > TRPO > DQN

Our results collectively show that pairwise models outperform pointwise and listwise models in terms of APFD for both sampled and full datasets. While the highest APFD score is achieved by the best model (PA-A2C) using the full dataset, the average APFD across various configurations is higher with the sampled dataset than with the full dataset. Moreover, as shown in Figure 3, utilizing the sampled dataset significantly reduces the training time of pairwise models; its maximum

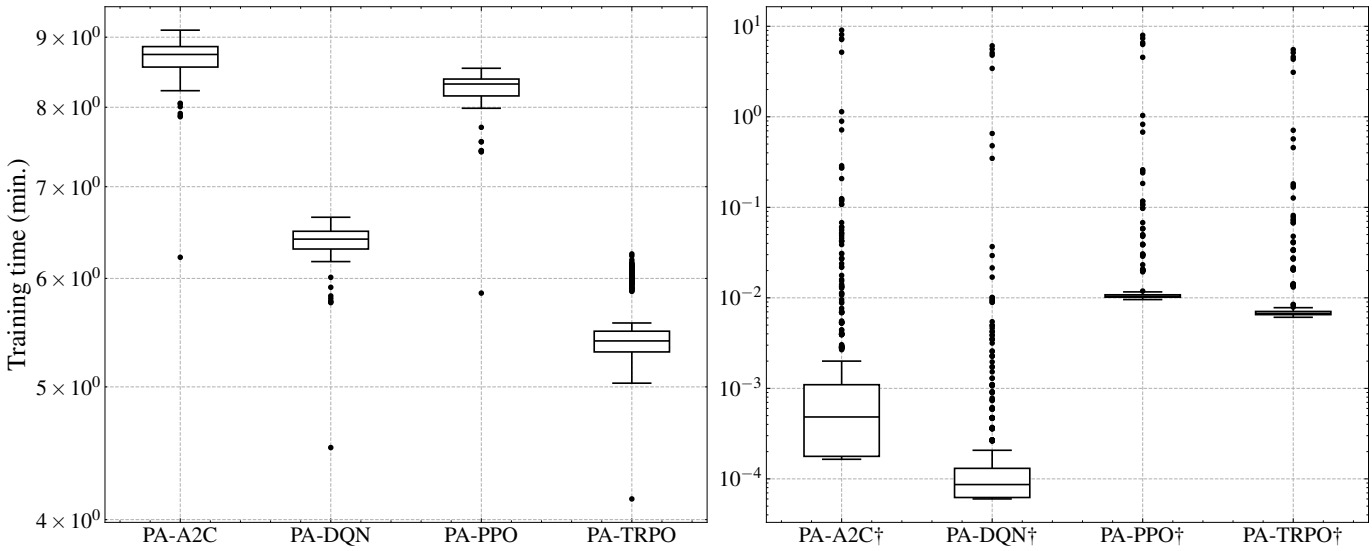

Fig. 3. Training time for pairwise configurations across different datasets

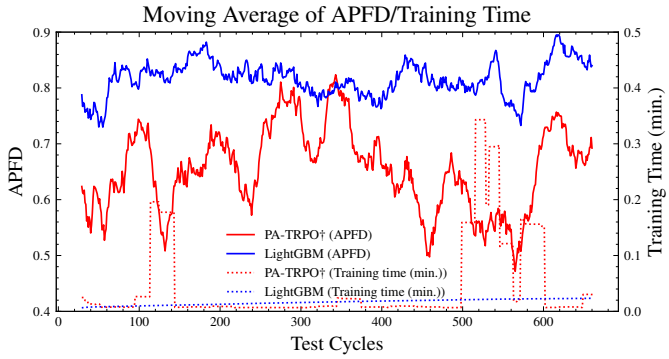

Fig. 4. 30-cycle moving average of APFD and Training Time for PA-TRPO† and LightGBM

TABLE II
WELCH'S T-TEST RESULTS AND COMMON LANGUAGE EFFECT SIZE BETWEEN ML-MODELS AND BASELINES.

Baseline	Model	p-val	CLES
Lexical Similarity	PA-A2C	0.000	0.738
	LightGBM	0.000	0.801
Test Coverage of Modified Code	PA-A2C	0.028	0.509
	LightGBM	0.000	0.598
Max Execution Time (90 days)	PA-A2C	0.692	0.533
	LightGBM	0.000	0.645
Max Execution Time (30 days)	PA-A2C	0.876	0.528
	LightGBM	0.000	0.642
Comprehensive	PA-A2C	0.000	0.678
	LightGBM	0.000	0.750

training time remains below 0.1 minutes, excluding outliers, whereas the worst-case training time exceeds 9 minutes when using the full dataset. Therefore, we conclude that using the sampled dataset for the pairwise model could be a promising approach in our scenario, as it shows comparable or better performance while significantly reducing training time.

B. Performance of SL Algorithms

Table I also contains the evaluation metrics from the tree-based ensemble models with RL approaches. LightGBM not only emerges as the top performer among the ensemble models, but also outperforms all RL-based TCP techniques. Notably, boosting algorithms (CatBoost, LightGBM, and XGBoost) consistently outperform the bagging algorithm, Random Forest. Further, both its training and prediction time are significantly shorter than RL-based TCP techniques. Figure 4 shows the 30-cycle moving average of APFD and training time reported by PA-TRPO†, the best-performing RL-based TCP with the sampled dataset, and the LightGBM, the best-performing SL-based TCP. Due to the increasing training data, the training time of LightGBM also increases but overall it remains very low. PA-TRPO†, on the other hand, shows high peaks due to random sampling used by our incremental learning setting.

In terms of MRR, LightGBM has an MRR of 0.133, meaning that it would produce the first failure after executing eight test cases on average ($1/0.133 \approxeq 7.52$). This is three fewer than PA-TRPO, whose MRR is 0.098: it means that PA-TRPO produces the first failure after executing 11 test cases on average ($1/0.098 \approxeq 10.2$). Based on these results, we conclude that, in the context of SAP HANA, retraining SL-based (especially boosting) models from scratch is both more effective and efficient than RL-based techniques.

C. Comparison with Baselines

In Table I, we found that the baselines generally performed similarly to the RL models but worse than the SL models in terms of APFD. To validate these findings, we additionally

report Welch's t-test and the Common Language Effect Size (CLES) for comparing the best-performing RL and SL models (PA-A2C and LightGBM, respectively) against the Baselines using the APFD metric.

Table II demonstrates that PA-A2C significantly outperformed when test cases were ordered by lexical similarity, test coverage of modified code, and a combined approach that considers four key features from the SL model. Specifically, the CLES values were 0.738, 0.509, and 0.678, indicating that PA-A2C outperformed the two baselines in 74%, 51%, and 68% of test cycles, respectively. However, certain baselines showed comparable average APFD scores to PA-A2C: Max Execution Time (90 days) at 0.743 and Max Execution Time (30 days) at 0.747. A Welch's t-test found no significant difference between these scores, suggesting that it is inconclusive whether the baseline or the RL model performs better overall. The statistical analysis results further reveal that the SL model, LightGBM, significantly outperformed all baselines, achieving a CLES between 0.598 and 0.801, indicating that it was more effective than all baselines in at least 60% of the cycles. It's worth noting that the baseline achieved by sorting Test Coverage of Modified Code outperformed LightGBM in terms of the MRR, but here we consider APFD as the primary measure of ranking accuracy in this analysis. Consequently, we can conclude that LightGBM, as an SL-based model, demonstrates the highest ranking accuracy.

V. Threats to Validity

Our evaluation dataset consists of eight months of testing data from SAP HANA, representing the longest available period for data collection at the time of our study. While this duration covers a diverse range of testing scenarios, a longer observation period could potentially reveal additional patterns and provide deeper insights into the effectiveness of TCP strategies.

Due to the high computational demands of RL training, we conducted our ML model training and evaluation on an Apple M2 Pro system rather than SAP's server infrastructure, where actual CI testing is performed. While our comparative analysis remains valid, the absolute training and prediction times may not directly reflect those in a production environment.

Our feature engineering approach did not explicitly account for flaky test behavior. Although final test verdicts in the actual testing environment were adjusted by aggregating outcomes across multiple executions, the failure rate and transition rate metrics used in our study treated all failures uniformly, which may have led to inflated values for tests affected by flakiness.

The choice of evaluation metrics for TCP can vary based on the specific goals of the testing process. To ensure a comprehensive assessment, we reported effectiveness results using multiple metrics. However, our statistical analyses and performance evaluation primarily focused on APFD, which is widely recognized for assessing TCP effectiveness.

VI. Conclusion

We report an evaluation of both Reinforcement Learning (RL) and Supervised Learning (SL) based Test Case Prioritization (TCP) techniques on real-world failure data collected from testing of SAP HANA. Our objective is to evaluate and compare the performance, as well as the cost, of these machine learning-based TCP techniques that can adapt to dynamic environments in the context of Continuous Integration for large-scale industrial projects.

Our evaluation considers three ranking models for learning-to-rank machine learning models, combined with seven RL algorithms, yielding 14 configurations on two datasets, totaling 28 configurations. Additionally, we evaluated four SL learning-to-rank models and five non-ML-based baselines, resulting in a total of 37 configurations. Our evaluation dataset is test execution history collected over a period of eight months, consisting of 647 test cycles.

The results indicate that, among the RL models, PA-A2C achieved the highest performance but required considerable training time, while PA-TRPO with a sampled dataset offered a balanced trade-off between performance and training efficiency. Furthermore, we also found that the best-performing RL model demonstrated comparable or even superior performance to non-ML baselines. However, the SL model LightGBM consistently outperformed both RL models and baselines in terms of effectiveness and efficiency, even with complete retraining at each test cycle. While RL supports incremental learning, it demands substantial training time and still underperforms in accuracy compared to SL, even when using data sampling (which reduces training time but yields limited accuracy improvements). This suggests that, even in our large-scale industrial setting, fully retraining an SL model for each cycle proves to be a more effective and efficient approach, while providing superior TCP performance and cost efficiency compared to RL.

References

[1] S. Yoo and M. Harman, "Regression testing minimization, selection and prioritization: a survey," *Software Testing, Verification and Reliability*, vol. 22, no. 2, p. 67–120, Mar. 2012. [Online]. Available: http://dx.doi.org/10.1002/stvr.430

[2] I. Bouzenia and M. Pradel, "Resource usage and optimization opportunities in workflows of github actions," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ser. ICSE '24. ACM, Feb. 2024. [Online]. Available: http://dx.doi.org/10.1145/3597503.3623303

[3] T. Bach, A. Andrzejak, C. Seo, C. Bierstedt *et al.*, "Testing very large database management systems: The case of sap hana," *Datenbank-Spektrum*, Nov. 2022.

[4] S. Kraft and C. Heer, "Cloud-native continuous integration system for large enterprise software projects," Apr. 2023, cI/CD Industry Workshop (CCIW).

[5] Y. Lou, J. Chen, L. Zhang, and D. Hao, *A Survey on Regression Test-Case Prioritization*. Elsevier, 2019, p. 1–46. [Online]. Available: http://dx.doi.org/10.1016/bs.adcom.2018.10.001

[6] J.-M. Kim and A. Porter, "A history-based test prioritization technique for regression testing in resource constrained environments," in *Proceedings of the 24th international conference on Software engineering - ICSE '02*, ser. ICSE '02. ACM Press, 2002. [Online]. Available: http://dx.doi.org/10.1145/581339.581357

[7] B. Busjaeger and T. Xie, "Learning for test prioritization: an industrial case study," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE'16. ACM, Nov. 2016. [Online]. Available: http://dx.doi.org/10.1145/2950290.2983954

[8] S. Yoo, R. Nilsson, and M. Harman, "Faster fault finding at google using multi objective regression test optimisation," in *8th European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE'11), Szeged, Hungary*, vol. 102, 2011.

[9] S. Elbaum, G. Rothermel, and J. Penix, "Techniques for improving regression testing in continuous integration development environments," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. SIGSOFT/FSE'14. ACM, Nov. 2014. [Online]. Available: http://dx.doi.org/10.1145/2635868.2635910

[10] A. Srivastava and J. Thiagarajan, "Effectively prioritizing tests in development environment," *ACM SIGSOFT Software Engineering Notes*, vol. 27, no. 4, p. 97–106, Jul. 2002. [Online]. Available: http://dx.doi.org/10.1145/566171.566187

[11] R. Carlson, H. Do, and A. Denton, "A clustering approach to improving test case prioritization: An industrial case study," in *2011 27th IEEE International Conference on Software Maintenance (ICSM)*. IEEE, Sep. 2011. [Online]. Available: http://dx.doi.org/10.1109/ICSM.2011.6080805

[12] J. Czerwonka, R. Das, N. Nagappan, A. Tarvo, and A. Teterev, "Crane: Failure prediction, change analysis and test prioritization in practice – experiences from windows," in *2011 Fourth IEEE International Conference on Software Testing, Verification and Validation*. IEEE, Mar. 2011. [Online]. Available: http://dx.doi.org/10.1109/ICST.2011.24

[13] R. Pan, M. Bagherzadeh, T. A. Ghaleb, and L. Briand, "Test case selection and prioritization using machine learning: a systematic literature review," *Empirical Software Engineering*, vol. 27, no. 2, p. 29, 2022.

[14] Y. Zhao, D. Hao, and L. Zhang, "Revisiting machine learning based test case prioritization for continuous integration," in *2023 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, Oct. 2023. [Online]. Available: http://dx.doi.org/10.1109/ICSME58846.2023.00032

[15] M. Bagherzadeh, N. Kahani, and L. Briand, "Reinforcement learning for test case prioritization," *IEEE Transactions on Software Engineering*, vol. 48, no. 8, p. 2836–2856, Aug. 2022. [Online]. Available: http://dx.doi.org/10.1109/TSE.2021.3070549

[16] L. Prokhorenkova, G. Gusev, A. Vorobev, A. V. Dorogush, and A. Gulin, "Catboost: unbiased boosting with categorical features," *Advances in neural information processing systems*, vol. 31, 2018.

[17] T. Chen and C. Guestrin, "Xgboost: A scalable tree boosting system," in *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining*, 2016, pp. 785–794.

[18] G. Ke, Q. Meng, T. Finley, T. Wang, W. Chen, W. Ma, Q. Ye, and T.-Y. Liu, "Lightgbm: A highly efficient gradient boosting decision tree," *Advances in neural information processing systems*, vol. 30, 2017.

[19] T. K. Ho, "Random decision forests," in *Proceedings of 3rd International Conference on Document Analysis and Recognition*, ser. ICDAR-95. IEEE Comput. Soc. Press, 1995. [Online]. Available: http://dx.doi.org/10.1109/ICDAR.1995.598994

[20] G. Rothermel, R. Untch, C. Chu, and M. Harrold, "Prioritizing test cases for regression testing," *IEEE Transactions on Software Engineering*, vol. 27, no. 10, p. 929–948, 2001. [Online]. Available: http://dx.doi.org/10.1109/32.962562

[21] A. Srivastava and J. Thiagarajan, "Effectively prioritizing tests in development environment," in *Proceedings of the 2002 ACM SIGSOFT international symposium on Software testing and analysis*, 2002, pp. 97–106.

[22] T.-Y. Liu, "Learning to rank for information retrieval," *Foundations and Trends® in Information Retrieval*, vol. 3, no. 3, p. 225–331, 2007. [Online]. Available: http://dx.doi.org/10.1561/1500000016

[23] 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 *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ser. ICSE '20. ACM, Jun. 2020. [Online]. Available: http://dx.doi.org/10.1145/3377811.3380369

[24] A. S. Yaraghi, M. Bagherzadeh, N. Kahani, and L. C. Briand, "Scalable and accurate test case prioritization in continuous integration contexts," *IEEE Transactions on Software Engineering*, vol. 49, no. 4, p. 1615–1639, Apr. 2023. [Online]. Available: http://dx.doi.org/10.1109/TSE.2022.3184842

[25] J. Chen, Y. Lou, L. Zhang, J. Zhou, X. Wang, D. Hao, and L. Zhang, "Optimizing test prioritization via test distribution analysis," in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE '18. ACM, Oct. 2018. [Online]. Available: http://dx.doi.org/10.1145/3236024.3236053

[26] A. Beszedes, T. Gergely, L. Schrettner, J. Jasz, L. Lango, and T. Gyimothy, "Code coverage-based regression test selection and prioritization in webkit," in *2012 28th IEEE International Conference on Software Maintenance (ICSM)*. IEEE, Sep. 2012. [Online]. Available: http://dx.doi.org/10.1109/ICSM.2012.6405252

[27] D. Elsner, F. Hauer, A. Pretschner, and S. Reimer, "Empirically evaluating readily available information for regression test optimization in continuous integration," in *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA '21. ACM, Jul. 2021. [Online]. Available: http://dx.doi.org/10.1145/3460319.3464834

[28] R. K. Saha, L. Zhang, S. Khurshid, and D. E. Perry, "An information retrieval approach for regression test prioritization based on program changes," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*. IEEE, May 2015. [Online]. Available: http://dx.doi.org/10.1109/ICSE.2015.47

[29] A. Raffin, A. Hill, A. Gleave, A. Kanervisto, M. Ernestus, and N. Dormann, "Stable-baselines3: Reliable reinforcement learning implementations," *Journal of Machine Learning Research*, vol. 22, no. 268, pp. 1–8, 2021. [Online]. Available: http://jmlr.org/papers/v22/20-1364.html

[30] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, "Openai gym," *arXiv preprint arXiv:1606.01540*, 2016.

[31] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu, "Asynchronous methods for deep reinforcement learning," in *International conference on machine learning*. PMLR, 2016, pp. 1928–1937.

[32] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, "Playing atari with deep reinforcement learning," *arXiv preprint arXiv:1312.5602*, 2013.

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

[34] T. Haarnoja, A. Zhou, P. Abbeel, and S. Levine, "Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor," in *International conference on machine learning*. PMLR, 2018, pp. 1861–1870.

[35] S. Fujimoto, H. Hoof, and D. Meger, "Addressing function approximation error in actor-critic methods," in *International conference on machine learning*. PMLR, 2018, pp. 1587–1596.

[36] J. Schulman, S. Levine, P. Abbeel, M. Jordan, and P. Moritz, "Trust region policy optimization," in *International conference on machine learning*. PMLR, 2015, pp. 1889–1897.

[37] G. Ke, Q. Meng, T. Finley, T. Wang, W. Chen, W. Ma, Q. Ye, and T.-Y. Liu, "Lightgbm: A highly efficient gradient boosting decision tree," *Advances in neural information processing systems*, vol. 30, 2017.

[38] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg *et al.*, "Scikit-learn: Machine learning in python," *the Journal of machine Learning research*, vol. 12, pp. 2825–2830, 2011.

[39] Q. Wu, C. J. Burges, K. M. Svore, and J. Gao, "Adapting boosting for information retrieval measures," *Information Retrieval*, vol. 13, pp. 254–270, 2010.

[40] C. Burges, R. Ragno, and Q. Le, "Learning to rank with nonsmooth cost functions," *Advances in neural information processing systems*, vol. 19, 2006.

[41] K. O. McGraw and S. P. Wong, "A common language effect size statistic." *Psychological bulletin*, vol. 111, no. 2, p. 361, 1992.

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

[43] P. A. Games and J. F. Howell, "Pairwise multiple comparison procedures with unequal n's and/or variances: a monte carlo study," *Journal of Educational Statistics*, vol. 1, no. 2, pp. 113–125, 1976.

[44] M. S. Bartlett, "Properties of sufficiency and statistical tests," *Proceedings of the Royal Society of London. Series A-Mathematical and Physical Sciences*, vol. 160, no. 901, pp. 268–282, 1937.