Comparative study of machine learning test case prioritization for continuous integration testing

Dusica Marijan

Accepted: 6 July 2023 / Published online: 28 July 2023
© The Author(s), under exclusive licence to Springer Science+Business Media, LLC, part of Springer Nature 2023

Abstract
There is a growing body of research indicating the potential of machine learning to tackle complex software testing challenges. One such challenge pertains to continuous integration testing, which is highly time-constrained, and generates a large amount of data coming from iterative code commits and test runs. In such a setting, we can use plentiful test data for training machine learning predictors to identify test cases able to speed up the detection of regression bugs introduced during code integration. However, different machine learning models can have different fault prediction performance depending on the context and the parameters of continuous integration testing, for example, variable time budget available for continuous integration cycles, or the size of test execution history used for learning to prioritize failing test cases. Existing studies on test case prioritization rarely study both of these factors, which are essential for the continuous integration practice. In this study, we perform a comprehensive comparison of the fault prediction performance of machine learning approaches that have shown the best performance on test case prioritization tasks in the literature. We evaluate the accuracy of the classifiers in predicting fault-detecting tests for different values of the continuous integration time budget and with different lengths of test history used for training the classifiers. In evaluation, we use real-world and augmented industrial datasets from a continuous integration practice. The results show that different machine learning models have different performance for different size of test history used for model training and for different time budgets available for test case execution. Our results imply that machine learning approaches for test prioritization in continuous integration testing should be carefully configured to achieve optimal performance.

Keywords Machine learning · Neural networks · Support vector regression · Gradient boosting · Learning to rank · Continuous integration · Software testing · Regression testing · Test prioritization · Test selection · Test optimization

✉ Dusica Marijan
dusica@simula.no

1 Simula Research Laboratory, Oslo, Norway
1 Introduction

Continuous integration (CI) is an agile software development practice where software is released frequently following frequent code changes. Each change needs to be verified before a new change can be made and a new version of the code released. This process runs in CI cycles, also called builds or commits. Typically, software testing runs iteratively and successively as part of continuous code integration. Each code integration is followed by an integration testing iteration, which aims to verify that individual code components, when combined, are working together, and which is typically extensive, to prevent breaking a build. CI testing requires a short turnaround between starting test execution and detecting faulty regressions (indicating that the existing functionality has been adversely affected), to enable fast feedback. This entails a short time budget allocated to integration testing, which denotes the amount of time available for testing the code changes introduced in the latest commit. Short time budget requires testing in CI to be time-efficient Niu et al. (2018); Savor et al. (2016); Parnin et al. (2017); Marijan et al. (2019); Marijan and Liaaen (2018); Marijan et al. (2018). As a response to this challenge, researchers have proposed various test selection, minimization, and prioritization Shi et al. (2019); Ali et al. (2020); Rothermel et al. (2001); Marijan (2015); Marijan and Liaaen (2017); Marijan et al. (2017) approaches. In this work, we specifically focus on test prioritization (TP). TP consists in ordering test cases that are more effective in detecting faults to execute sooner. In this way, we ensure that the most important test cases are executed in a short time budget. However, in dynamic CI environments with frequent code changes, a time budget can vary across different CI cycles. Therefore, an efficient TP approach needs to adapt to varying time constraints across CI cycles.

Furthermore, given that testing runs frequently in CI generating a large volume of test information, researchers have proposed history-based TP approaches. These approaches use historical test execution information to speed up the detection of regression faults introduced by developers. Hemmati et al. (2017) suggests that history-based TP is an effective approach for rapid release software, while Srikanth et al. (2016) reports that using historical test failure information is a good indicator for test prioritization. However, history-based TP has its challenges. One common challenge is to decide how old historical information to use in TP. On the one hand, using too old history may capture old (irrelevant) failures which have been fixed and thus are not indicative of new failures. On the other hand, using too recent history may omit some relevant failures. To deal with this challenge, Elbaum et al. (2014) introduced the notion of time windows, to capture how recently tests were executed and failures exposed, which is further used for test selection in pre-submit and post-submit testing.

1.1 Motivating example

We illustrate one example of CI testing, describing daily practices and challenges of our industrial collaborator in the domain of testing configurable communication software in CI, shown in Fig. 1. The project involves a product line of conferencing systems, where each product has up to a thousand test cases, totaling several thousands of test cases for the whole product line. Following a standard practice, code changes committed by developers are regression tested before they can be deployed to production. Regression testing implies re-running tests to ensure that the tested code works as intended after changes. Several hundreds of changes
made on a daily basis trigger the execution of several thousands of test cases. Change impact analysis (based on traceability links between code and test cases, where the links are created and maintained automatically) is run to select the test cases impacted by the change. However, all impacted test cases cannot fit the available time budget and are therefore usually not executed all. Moreover, not all impacted test cases are equally useful in detecting faults. If test engineers were to manually select a subset of tests produced by change impact analysis that they believe have the highest chance of detecting faults, such a process would be highly time-inefficient. Thus, test engineers have applied automated regression TP, as an established approach to improve the effectiveness of regression testing in CI. Specifically, given code changes and test execution history, the applied regression TP approach Marijan et al. (2013) computes an ordered set of test cases that are impacted by the code changes and that are of the highest historical fault detection ability. This ensures that the most relevant test cases have been executed in case that the test execution budget is reduced. With this approach, test engineers can detect up to 30% more regression faults compared to manual test selection guided by the tester’s expertise, for the same time budget Marijan et al. (2013). However, this approach is not well suited for processing a large set of historical test execution data. Following a recent research direction of using machine learning (ML) for software testing, the goal of test engineers has been to develop an ML approach for TP that will be both time-efficient and have a high fault-detection efficiency as more test data becomes available.

While developing an ML-based test prioritization approach addressing the needs of our industrial partner, we observed that different ML models can have different fault-detection performance depending on the CI testing context. This was especially the case when we used ML models in CI cycles with different time budgets (budget for testing) and when we used different sizes of test execution history for ML model learning. Therefore, we systematically studied how do these two parameters affect the fault-prediction performance of ML-based test prioritization models.

### 1.2 Contributions

In this paper, we report the experimental results of the systematic comparison of four best-performing ML approaches reported in the literature on the task of test case fault prediction: support-vector machines (SVM), artificial neural networks (ANN), gradient boosting decision trees (GBDT), and LambdaRank using NN. In addition, we compare the performance of ML-based approaches against two heuristic approaches: history-based TP approach ROCKET Marijan et al. (2013) and Random test selection. We run the
experiments on three industrial data sets from the CI practice: Cisco, ABB, and Google. In addition, we use two additional datasets which were augmented to address the class imbalance problem (too few failing test cases) in the industrial datasets, thus improving the quality of training data for the ML approaches. We report the results in terms of fault-detection effectiveness and time-effectiveness when there is a varying time budget available for testing and when there is a different length of test history used for model learning.

In summary, our work makes the following contributions:

- Systematic analysis of how the size of test history affects fault-prediction effectiveness of learning-based test case prioritization. CI testing produces voluminous test history which is used to learn how to prioritize test cases. Understanding how much old test history is more predictive of failing test cases can help tune ML-based test case prioritization approaches. Our results indicate that the optimal size of test history used for learning to prioritize tests varies across datasets depending on the number of test cycles in a dataset. When comparing the performance of different ML models for TP relative to the size of test history, the neural network model showed to be the least sensitive to using older test history.

- Systematic evaluation of the effectiveness of ML-based test case prioritization approaches relative to variable time budget across CI cycles. A variable time budget is inherent to CI testing, where the test budget changes depending on how much time is available for the CI cycle. Understanding how the variation of the time budget affects test case prioritization can improve the effectiveness of CI testing. Our results show that the fault-prediction performance of ML models greatly varies depending on how much time is available for test execution.

- Systematic comparison of the best-performing ML-based test case prioritization approaches reported in the literature, evaluated on three+two industrial datasets. On these datasets, our results indicate that the best-performing approach in terms of fault-detection effectiveness, time to detect the first fault, time to detect the last fault, and training time is LambdaRank using NN. The results further show that the best performing approach in terms of ranking time is a neural network. Overall, our results imply that machine learning approaches for test prioritization in continuous integration testing should be carefully configured to achieve optimal performance.

The paper is structured as follows. In Sect. 2, we review related work. In Sect. 3, we describe learning-based test case prioritization and present the four ML-based test prioritization approaches evaluated in this study. Section 4 describes the experimental evaluation, while Sect. 5 presents the experimental results. We discuss the key findings of the study and conclude the paper in Sect. 6.

2 Related work

Recent studies have used ML for the problem of test case prioritization. Machalica uses a boosted decision tree approach to learn a classifier for predicting a probability of a test case failing based on code changes and a subset of test cases Machalica et al. (2019). The approach was shown to reduce the testing cost by a factor of two while ensuring that over 95% of individual test failures and detected. Chen proposes another predictive test prioritization approach based on XGBoost Chen et al. (2018). It studies test case distribution
analysis evaluating the fault detection capability of actual regression testing. The approach has been used in practice and has been shown to significantly reduce testing costs. Motivated by the success of these two approaches based on gradient boosting, we selected the GBDT as an evaluation candidate for our study.

Busjaeger proposes a test case prioritization approach based on support vector machine (SVM) Busjaeger and Xie (2016) to learn a binary classifier to order test cases based on historical information. The approach has been shown to outperform non-ML-based test case prioritization approaches in terms of fault-detection effectiveness. Lachmann Lachmann et al. (2016) uses SVM-Rank to prioritize test cases using test case failure information. The evaluation shows that SVM-based approach to test case prioritization outperforms manual approaches by experts. Grano uses SVM and random forest (RF) to build a regression predictive model for assessing test branch coverage Grano et al. (2018) for the purpose of efficient test case generation for CI testing. The experimental results have shown good fault prediction accuracy of SVM for test case prioritization; therefore, we selected SVM as an evaluation candidate in our study.

Several test case prioritization approaches have been proposed using different forms of neural networks, such as Bayesian network Mirarab and Tahvildari (2008), NN Mahdieh et al. (2020), ANN Jahan et al. (2019), and RNN Hasnain et al. (2019). Specifically, Mirarab and Tahvildari (2008) integrates a feedback mechanism and a change information gathering strategy to estimate the probability of a test case to find bugs. The approach has been shown to enable early fault detection. Mahdieh et al. (2020) prioritizes test cases using a NN approach and the fault-proneness distribution of different code areas. The approach has been shown to improve the effectiveness of coverage-based test case prioritization. Jahan et al. (2019) uses the combination of test case complexity information and software modification information to train an ANN to enable early fault detection. The approach has been shown to improve fault detection effectiveness. Hasnain et al. (2019) proposes a gated recurrent unit trained on the time series throughput information to perform regression testing of web services. The results have shown good fault prediction performance. Following the good fault-prediction performance of these studies, we included the ANN approach in our evaluation study.

There are studies using reinforcement learning (RL) for test case prioritization, which focus on maximizing a reward when failing test cases are prioritized higher Shi et al. (2020) or on using simpler ML models for RL policy design Rosenbauer et al. (2020). Lima proposes a multi-armed bandit (MAB) approach to test case prioritization in continuous integration Lima and Vergilio (2020), which showed to outperform the RL approach in terms of fault detection. However, we experimented with these approaches for test prioritization, and they showed to be computationally expensive Sharif et al. (2021). Furthermore, Bertolino et al. (2020) conducts an extensive experimental study comparing RL against supervised learning for test case prioritization and concludes that the RL approach is less efficient on this specific task. Because of our experience with RL and the experience reported by Bertolino, we did not select the RL and MAB approaches for our evaluation study, as our goal is to build a fast-running test case prioritization approach that can satisfy strict time constraints of short CI cycles. Even if the prediction of a learned model is fast, if we need to retrain the model often, it is desirable that the retraining is fast too.

In the same study Bertolino et al. (2020), Bertolino reports the best-performing ML approach to test case prioritization in terms of fault-detection effectiveness are MART and LambdaMART. Motivated by this finding, we include LambdaRank in our evaluation study. LambdaRank is from the same family of learning to rank algorithms as MART, and we include it instead of MART, as MART is based on gradient boosted decision trees which we have already included in our study.
### 3 Learning based test prioritization

In CI development practices, testing is time-constrained and produces voluminous test history $H$, as CI cycles run fast and frequently. The test history $H$ contains test execution information for all CI cycles $C_i$, denoted as cycle history, where $i = 1...n$ and $n$ is the number of CI cycles (test executions). Each cycle history consists of a test suite $T = \{T_1, T_2, ..., T_n\}$ run in that cycle and the time budget of the cycle $B$. Each $T_i$ has its execution time $t_i$ and $n$ historical execution results $\{R_{i,1}, R_{i,2}, ..., R_{i,n}\}$, where $R \in \{0, 1\}$ denotes a test pass or fail. Although it is possible that the value of $t_i$ varies across different cycles, in this work, we assume that $t_i$ is the average execution time of a test case across its CI cycles, as done in Marijan et al. (2013).

Given $H$, collected in runs in previous CI cycles, the goal of the learning-based test case prioritization is to predict which test cases will be effective in detecting faults in the current CI cycle $C_{n+1}$, ranked according to their fault detection effectiveness. In addition to fault detection effectiveness, some approaches use test execution time $t$ as another prioritization criteria, which can be combined together Marijan et al. (2013) to ensure that failing test cases are ordered higher, and among the failing test cases, those that execute faster are ordered higher.

In history-based test case prioritization, historical test failure records may be weighted, such that the highest failure weight corresponds to the failure exposed in the most recent test case execution and the failure in every precedent test execution is weighted lower. This ensures that the test cases that failed in the most recent run will be ordered higher (thus executed first), followed by a number of “older” failed test cases, depending on the available time budget $B$. Such “older” failed test cases are execution candidates as well, because tests can be flipping from fail to pass to fail again, as illustrated in Fig. 2. In case of ties, i.e., two or more test cases have the same failure probability, test cases should be ranked in the order of the shortest execution time $t$. We can define the problem of learning-based regression test prioritization as follows:

For a test case $T_j$ belonging to a regression test suite $T = \{T_1, T_2, ..., T_n\}$, the goal of learning is to find a function $g : T \rightarrow C$, mapping the test case $T_j$ to a class $C_i$ (test rank) belonging to $C = \{C_1, C_2, ..., C_m\}$, where $T_2$ is ranked higher than $T_1$ if $g(T_2) > g(T_1)$, $m$ is the number of test ranks. In binary classification $C \in \{0, 1\}$ Marijan et al. (2013).

![Test runs = CI cycles](image-url)

**Fig. 2** Test history consisting of 15 CI cycles. Cycle 1 is the most recent and has the highest weight, while cycle 15 is the oldest and has the lowest weight. Test cases can change execution results between pass and fail in consecutive executions (CI cycles). Red: fail, green: pass, grey: inconclusive. In this work, we only deal with pass and fail test results.
3.1 Selection of ML approaches for test prioritization in CI

As discussed in the related work, there are many ML approaches for test case prioritization. However, as we are interested in improving the efficiency of test prioritization in the CI practice, which is highly time-constrained, in our industrial case study, we were looking for a time-efficient ML approach that can serve the need of generating prioritized test suites quickly. For example, we have previously experimented with RL for test case prioritization in comparison with the NN approach on four industrial datasets Sharif et al. (2021) and have found the total runtime of the RL approach to be around 50 times higher than the runtime of the NN approach. This is consistent with the results reported by Bertolino et al. (2020). Therefore, we excluded RL approaches from this comparative study. Driven by the requirement to build a fast-running ML approach to test prioritization, we implemented four simpler types of classifiers for learning to prioritize regression tests, which have previously showed good fault detection performance, as discussed in the related work. The classifiers are learned on historical test execution results generated throughout several months of testing. Next, we describe the four classifiers, i.e., support vector machine (SVM) classifier, artificial neural network (ANN) classifier, gradient boosted decision tree (GBDT) classifier, and LambdaRank with NN (LRN) classifier.

3.2 SVM model

SVM is a binary classifier by nature, which means that it can separate only two classes. To extend SVM for a multiclass classification problem, we employed two strategies: One-vs-All (OVA) and One-vs-One (OVO).

For the sake of illustrating the method, let us assume there are five ranks of regression tests from $C_1$ to $C_5$, where rank $C_1$ has the highest priority, and rank $C_5$ the lowest priority during test execution. Consequently, we consider regression test prioritization a multiclass classification problem, with one class for each of the five ranks.

OVO strategy, in this case, uses $5 \times (5 - 1)/2 = 10$ binary SVM classifiers, with one classifier for every pair of distinct classes from the training set, which the classifier needs to learn to distinguish. During training, a classifier $SVM_{ij}$ uses the samples from the $i$-th class as positive and the samples from the $j$-th class as negative. We used Max-Wins voting strategy to determine instance classification. Specifically, for a new sample to be classified, if $SVM_{ij}$ finds it in the $i$-th class, then the vote for the $i$-th class is increased, and otherwise, the vote for the $j$-th class is increased. Finally, after all classifiers have voted, a new sample is assigned to the class with the most votes. OVO strategy for a 5-class classification problem is illustrated in Fig. 3.

OVA strategy uses 5 binary SVMs for a 5-class classification problem, one per class. Specifically, during training, the $i$-th SVM uses the samples from the $i$-th class as positive, and the samples from all other classes as negative. Each binary classifier outputs a confidence value. To determine instance classification, we used Winner-Takes-All strategy. Specifically, when there is a new sample to be classified, all classifiers produce a confidence value for that sample. Next, the sample is assigned the class for which there is the highest output value produced. OVA strategy for a 5-class classification problem is illustrated in Fig. 4.
3.3 ANN model

To address a multiclass problem of regression test prioritization, we built a multiclass NN, which deals with a multiclass classification by reducing to a set of binary classification problems. To this end, we employed OVA and OVO transformation strategies. During the training phase, training data for binary classifiers is divided into sets, and during the prediction phase, outputs produced by binary classifiers are combined together using a decision function to determine an instance classification. Figure 5 illustrates the OVA strategy for a 5-class regression test classification.

We implemented NN consisting of input, output, and hidden layers. The input consists of input features, such as test execution status in previous CI cycles \( R_i, R_{i2}, \ldots, R_{in} \), where \( R \in \{0, 1\} \) means a test pass or fail, average test execution time \( t_i \), a distance of a failure to the current execution (because we consider recent failures more critical), change in the pass/fail execution status of a test \( R \). For the sake of illustration, we assume the output layer consists of five output neurons, one for each class. The number of hidden layers and their size is determined by an exhaustive hyperparameter tuning using grid search. In each hidden layer, neurons first receive weighted inputs from a previous layer and compute outputs using an activation function and then pass the results to the next layer. This process repeats until it reaches the output layer.

3.4 GBDT model

Our GBDT model is based on boosting. Boosting works in iterations by combining weak predictions produced in each iteration, which are only partially accurate, into a strong prediction. Specifically, we use a learning-to-rank algorithm, RankBoost, which is well suited
for ranking learning problems. We designed our GBDT model to rank a regression test suite $T$ by learning test ranks $C_i$ from historical test executions and ranks. Each weak learner generates a weak test order, which is considered intermediate and is adjusted in the following boosting cycle. The final test case ranking produced by the GBDT model is a weighted sum of all weak predictions. In our model, weak learners are decision trees. Each tree minimizes the error from the previous tree, and a newly added tree is fitted to the residuals from the previous tree through the loss function. The learning process is illustrated in Fig. 6.

### 3.5 LRN model

LamdaRank is a learning to rank method which defines ranking as a pairwise classification problem. We extended LamdaRank by representing hidden layers with a NN model. The input layer consists of test case features, such as historical test execution status (from previous CI cycles), average test execution time, the distance of a failure to the current execution, change in the pass/fail execution status of a test, which are fed to the hidden layers. The hidden layers compute a pair of test cases $T_i, T_j$ at each timestep. Using sigmoid function, two test cases are mapped to the learned probability that one test case should be ordered higher $P_{ij} = P(T_i > T_j) = \frac{1}{1+e^{-\sigma(s_i-s_j)}}$. The algorithm is illustrated in Fig. 7.
4 Experimental evaluation

The goal of the experimental study is to evaluate and compare the performance of four ML-based test case prioritization approaches discussed in Sect. 3 with the aim of answering the following research questions:

1. RQ1 Does the length of test execution history used for learning to prioritize test cases impact the fault-prediction performance of ML approaches?
2. RQ2 Which ML approach is more effective in predicting test cases with higher fault detection effectiveness, for a given time budget, and how do they compare to heuristic-based test case prioritization approaches?
3. RQ3 Which ML approach is more time-efficient in a test prioritization task, and how do they compare to heuristic-based test case prioritization approaches?

4.1 Experimental dataset

We perform experimental evaluation on three industrial datasets used for integration testing in CI: Cisco, ABB, and Google. Cisco dataset is used for testing video conferencing systems, provided by Cisco Systems. ABB dataset\(^1\) is used for testing painting robot software, provided by ABB robotics. Google dataset\(^2\) is from a large scale continuous testing infrastructure provided by Google Elbaum and Penix (2014). The datasets contain information about the number of test cases, the number of test executions (CI cycles) for each test case, and the historical fault-detection effectiveness of each test case in each execution as pass or fail. Test cases in the proprietary Cisco dataset are integration tests, used for testing the integration of different videoconferencing system components together in the final product. The tests are developed using the Cisco’s proprietary test case specification framework based on the .json format. Test cases in the ABB dataset are also integration testing tests, used for testing the interfaces between different robotic system modules and exposing faults when these modules are interacting with each other. The tests are developed using a proprietary test framework written in Python. The Google dataset contains information on a sample of over 3.5 million integration test case executions, gathered over a period of 30

\(^1\) https://bitbucket.org/HelgeS/atcs-data/src/master/
\(^2\) https://code.google.com/archive/p/google-shared-dataset-of-test-suite-results/
days, applied to a sample of Google product Elbaum et al. (2014). Test cases are developed using specialized XUnit-like frameworks, such as the Google C++ Testing Framework.

In addition, we use two augmented industrial datasets, AugCisco and AugABB, for the experiment answering RQ2 and RQ3. This is because the RQs aim to evaluate the effectiveness of prioritized test suites in enabling early fault detection. However, the industrial datasets have a low percentage of failures, i.e., they are imbalanced, which makes it a less challenging task to detect faults early than if they contained a high percentage of failing test cases. To manage this problem, known as the class imbalance problem in training, we used SMOTE Lemaître et al. (2017) to augment the industrial datasets by increasing the minority samples (failed test cases). Due to a large number of test executions in the Google dataset, we do not augment this dataset. We summarize the datasets in Table 1.

### 4.2 Evaluation baselines

We compare the ML models for test prioritization one against the other, as well as against the automated TP approach ROCKET Marijan et al. (2013) that has previously been shown to improve the effectiveness of a manual practice of test selection at Cisco, and the Random approach.

ROCKET prioritizes a set of test cases in CI testing based on historical test execution status and test execution duration. The basic principle of ROCKET is that given the statuses of test cases’ previous runs in successive CI cycles and their average execution time, the algorithm computes a priority value for each test case to maximize early fault detection. More information about ROCKET can be found here Marijan et al. (2013). We varied the length of the historical information used for prioritization by ROCKET from the most recent 20% to the whole test history size available, with an increment of 20%. Variable length of test execution history is not applicable to Random heuristic, because it orders test cases randomly, without considering their historical fault-detection effectiveness during test selection.

### 4.3 Evaluation metrics

We perform the comparison of the TP approaches in terms of the following metrics:

**APFD**: Measures the weighted average of the percentage of faults detected by a test suite. Given a prioritized test suite \( S = \{S_1, S_2, ..., S_n\} \), a set of faults detected by \( S \) as

| Dataset  | # tc | # test executions | % failed tc | # cycles |
|----------|------|-------------------|-------------|----------|
| Cisco    | 550  | 6050              | 0.43        | 110      |
| ABB      | 1488 | 149700            | 0.28        | 100      |
| Google   | 5507 | 12439910          | 0.01        | 2259     |
| AugCisco | 550  | 6050              | 11.00       | 110      |
| AugABB   | 1488 | 149700            | 10.00       | 100      |

3 https://code.google.com/p/googletest/
F = \{F_1, F_2, ..., F_m\}, and SF_i as the number of test cases executed in S before exposing fault F_i, then APFD = 100 * (1 - (SF_1 + SF_2 + ... + SF_m)/nm + 1/2n). The higher the APFD, the better the fault-detection performance. Proposed by Rothermel et al. (2001), this is the most commonly used metric for evaluating the effectiveness of test prioritization Khatibsyarbini et al. (2018).

**TDFT:** Measures the time to detect the first fault by a prioritized test suite. Given a prioritized test suite S = \{S_1, S_2, ..., S_n\}, a set of faults detected by S as F = \{F_1, F_2, ..., F_m\}, and a set of fault detection times for each F_i as T = \{T(F_1), T(F_2), ..., T(F_m)\}, then \(TDFT = T(F_i)\), where \(T(F_i) < T(F_j), \forall T(F_j) \in T\). This metric evaluates the capability of early fault detection.

**TDLF:** Measures the time to detect the last fault by a prioritized test suite. Given a prioritized test suite S = \{S_1, S_2, ..., S_n\}, a set of faults detected by S as F = \{F_1, F_2, ..., F_m\}, and a set of fault detection times for each F_i as T = \{T(F_1), T(F_2), ..., T(F_m)\}, then \(TDLF = T(F_i)\), where \(T(F_i) > T(F_j), \forall T(F_j) \in T\). This metric also evaluates the capability of early fault detection.

**TRAIN:** Measures the training time of an ML model for test prioritization.

**PART:** Measures the overall running time of a prioritization algorithm, i.e., ranking time.

### 4.4 Experimental setup

First, for the evaluated ML-based approaches, for the purpose of model learning, we used a varying length of test history for all three datasets (Cisco, ABB, and Google), from the most recent 20% (approximately corresponding to the most recent 20% of CI cycles) to the whole test history available, with an increment of 20%, which we denote as \(H1-H5\). Next, while performing TP, we study whether the fault-detection effectiveness of the learned ML model changes if older test history is used for model learning. The basic idea of ML is that more data yields better performance. However, in the case of CI testing, using more historical cycles for learning may or may not mean better prediction performance Marijan and Liaaen (2016), since some of the previous faults might have been fixed in previous CI cycles, and in that case, they are no longer good predictors of failing test cases. Next, we ran the 20 learned ML models for each of the three experimental test suites: Cisco, ABB, and Google to produce prioritized test suites. Next, we ran the two heuristic-based test prioritization approaches, ROCKET and Random, for all three datasets.

In the next part of the experiment, we selected the learned classifiers with the best size of test history used for model learning and produced the prioritized test suites. Next, we run the prioritized test suites to evaluate their fault-detection effectiveness using five varying values of the time budget (B1-B5). B5 corresponds to the average time required to run the whole test suite, and B1 corresponds to 20% of that same budget. The remaining time budgets increase from B1 to B5 with increments of 20%. By decreasing the time budget, we can assess the effectiveness of a test suite to detect failing test cases earlier, because a well-performing ML predictor would prioritize failing test cases higher. In addition to running the prioritized test suites on the Cisco, ABB, and Google datasets, we use the augmented datasets AugCisco and AugABB which have more failing test cases and thus allow us to better evaluate the ability of TP approaches to detect faults early for short time budgets.

In the final part of the experiment, we measured the time effectiveness of the TP approaches in terms of training time (for the ML approaches), ranking time, and time to...
detect the first and last fault and compared them with the heuristic-based TP approaches. We measure all the metrics on the Cisco, ABB, and Google datasets. In addition, we measure the time to detect the last fault also on the augmented datasets AugCisco and AugABB. The augmented datasets allow us to better evaluate the ability of TP approaches to detect faults early.

Training the ML models requires parameter tuning. Specifically, to achieve good performance of the GBDT model, we experimented with two hyperparameters: learning rate and n_estimators. The learning rate affects the rate of adding new trees to the model. For example, a lower learning rate usually gives a more generalized learner. However, a lower learning rate needs more time for model training, and it requires a higher number of trees. Many trees may lead to overfitting. Therefore, choosing an optimal learning rate and n_estimators is important for the good performance of the GBDT model. Similarly, the performance of the learned NN classifier is dependent on different parameters, such as the number of hidden layers and their sizes, activation function, and the number of epochs. To learn a well-performing classifier, we performed an exhaustive hyperparameter tuning. We trained several classification models, while varying the number of hidden layers and layer sizes for each layer. ReLu was used as the activation function for the hidden layers. Each network had 50 epochs, and the training process of each network was iterated ten times, while measuring the average mean square error (MSE) and standard deviation (SD) of MSE for all five networks. Finally, we chose the best-performing target 3-layer network with minimal MSE and SD.

5 Results and analysis

In this section, we first analyze the experimental results answering the research questions and then discuss the main threats to the validity of the reported results.

5.1 RQ1: Effect of test history size on fault-prediction performance

We show the fault-detection effectiveness of history-based TP approaches for different lengths of test history used for learning to prioritize in Fig. 8. Overall, our experimental results indicate that the fault-prediction performance of the history-based approaches for TP (both ML-based and ROCKET) varies depending on how much test execution history is used in learning to prioritize. Specifically, for the shortest length of test history (H1), all TP approaches achieve low performance. As the length of test history increases (H2), the fault-prediction performance of all TP approaches increases across all datasets. Increasing the length of test history further (H3) has a positive effect on all TP approaches for the Cisco and ABB datasets. However, for the Google dataset, we see a decrease in the performance for all TP approaches except ANN in H3. Increasing the length of test history further (H4) has a negative effect on all TP approaches across all datasets except ANN for the Google dataset. Overall, we see that there is less negative effect on ANN compared to other ML approaches. However, the results show that ROCKET is more negatively affected by using older test history than the ML approaches. This could be because learning approaches need more data to learn to prioritize, and therefore, the negative effect of too long history is lower. As we continue to increase the length of test history (H5), the performance of all approaches decreases, and more significantly for ROCKET than for the ML approaches. Among the ML approaches specifically, we see that the ANN approach is less sensitive to using older test history than other ML approaches. Also, the ANN approach showed to be
the most sensitive to using younger test history (for example, it has the worst performance out of all ML approaches for $H_3$ on the Cisco and ABB datasets).

The results indicate that the size of the test execution history used for learning to prioritize affects the fault-prediction performance of TP approaches. Specifically, for the Cisco and ABB experimental datasets, the optimal size of test history has shown to be $H_3$, which corresponds to 60% of test history. For the Google datasets, the optimal size of test history

Fig. 8 Performance of TP approaches in terms of APFD for different sizes of test history used for learning to prioritize ($H_1$-$H_5$) across three datasets: Cisco, ABB, and Google
in our experiment was 40%. This implies that the optimal size of test execution history decreases with the increase of test cycles. For example, the Google dataset has longer test history of 2259 cycles, while the Cisco and ABB datasets have only around 100 cycles. This may also mean that the optimal size of test history is dependent on the frequency of bug fixing and code commits, i.e., the frequency of CI cycles. For datasets with longer test history, less percentage of it should be used for test prioritization compared to the datasets with a shorter history. The results are summarized in Fig. 9 in terms of average APFD for different sizes of test history. A trend can be observed for all approaches except Random, showing how the size of test history affects average APFD.

**Summary**: The size of the test execution history used for learning to prioritize affects the fault-prediction performance of TP approaches. The optimal size of test execution history decreases with the increase of test cycles and may be dependent on the frequency of bug fixing and code commits, i.e., the frequency of CI cycles.

### 5.2 RQ2: Fault-detection effectiveness for different time budget

To answer this research question, we use the ML models with the best configuration of test history size, $H_3$ for the Cisco and ABB datasets and $H_2$ for the Google dataset. We compare the fault-detection performance of the four ML models and two heuristics in terms of APFD, relative to the time budget available for running prioritized test suites. The results are shown in Fig. 10. Different colors of the bars correspond to five different values of the time budget $B_1 - B_5$, starting from 20% of the average overall time required
to run a test suite, with increments of 20%, while the stacked columns represent the increment in APFD with respect to the previous time budget.

In addition to the regular datasets (Cisco, ABB, and Google), we compare the TP approaches on two augmented datasets (AugCisco, AugABB), since the regular datasets have a low percentage of failed test cases, in which case most of the failures could be detected during the first few time budgets, e.g., $B_1$, $B_2$, and $B_3$.

The results indicate that the LRN approach and ROCKET approach achieve similar performance on average. Both approaches perform better for longer time budgets, with LRN having a slightly higher APFD for longer time budgets compared to ROCKET, and ROCKET a slightly higher APFD for shorter time budgets compared to LRN for some datasets, e.g., Cisco and ABB. It is expected that a longer time budget enables higher fault detection, as there is more time available for testing, more test cases can be executed, and more faults detected. GBDT comes as the next best-performing approach, followed by SVM, on all datasets except Cisco. For this particular dataset, SVM slightly outperforms GBDT. ANN approach has the worst fault-detection performance for short time budgets out of all ML-based approaches. Its performance improves for larger time budgets. Furthermore, Random has the absolute worst fault-detection performance for short time budgets out of all evaluated approaches, while its fault-detection effectiveness improves for larger time budgets.

Finally, for the augmented datasets AugCisco and AugABB which have a higher percentage of failing test cases, for all ML approaches and ROCKET, as expected, there were fewer faults detected in the smaller ($B_1$, $B_2$) compared to the larger ($B_3$, $B_4$, $B_5$) time budgets, in comparison with the regular datasets (Cisco, ABB, and Google), which have a lower percentage of failing test cases.

To find if there are significant differences between the classifiers, we use a nonparametric hypothesis testing method, the Friedman test. The test compares the median values of classifiers to test if differences between the classifiers are statistically significant. We found some significant differences in rankings with the Friedman test. The ranking returned by the test is shown in Table 2, while and the $p$ – values are shown in Table 3.

| SVM | GBDT | ANN | LRN | ROCKET | Random |
|-----|------|-----|-----|--------|--------|
| Ranking | 4.03 | 2.95 | 5.01 | 1.33 | 1.68 | 5.98 |

For the visual comparison of multiple classifiers across multiple datasets, we use a critical difference (CD) graph Calvo and Santafé Rodrigo (2016). CD diagrams are useful for comparing outcomes of multiple treatments over multiple observations. We use them to show the ranking of classifiers based on average classification error, averaged across all datasets. This means that classifiers with a lower rank perform better. In our CD diagrams, the leftmost reported classifier performs the best. The horizontal line between two classifiers means they have no statistically significant difference in performance (for example, LRN and ROCKET in Fig. 11).

The CD diagram showing our results averaged across all five datasets is shown in Fig. 11. The leftmost shown approach LRN performs the best, and its results have no statistically significant difference with the ROCKET approach. The rightmost approach Random performs the worst.
5.3 RQ3: Time effectiveness

Time effectiveness is measured using four metrics: TDFF, TDLF, TRAIN, and PART. We measure TDFF on three datasets (Cisco, ABB, and Google) because the metric is not affected by a low percentage of failed test cases. However, TDLF is affected by a low percentage of failed test cases, and therefore, we measure this metric on two augmented datasets: AugCisco and AugABB. We report all the metrics in terms of the percentage of the time budget of a CI cycle.

In terms of the time to detect the first fault (TDFF), LRN has the best performance, which is comparable to SVM. The next best-performing approach is ROCKET, followed by GDBT. The ANN model has the worst ability to detect faults early out of all ML-based approaches. However, Random has the worst performance out of all evaluated approaches. Average results per dataset are shown in Fig. 12, and the critical difference graph for the

![Figure 11](image)

**Figure 11** Average error ranking of the evaluated classifiers in terms of APFD for different time budgets. The leftmost performs the best.
results averaged across all three datasets is shown in Fig. 13. The ranking returned by the test is shown in Table 4, while the \( p \) – values are shown in Table 3.

In terms of the time to detect the last fault (TDLF), we can see different performance for non-augmented and augmented datasets, as expected. In both cases, LRN shows to be a superior approach, followed by SVM and ROCKET which have comparable performance. The next best-performing approach is GBDT, followed by ANN. Random shows the worst performance. Average results per dataset are shown in Fig. 14, and the critical difference graph for the results of six evaluated TP approaches averaged across all five datasets is shown in Fig. 15. The ranking returned by the test is shown in Table 5, while \( p \) – values are shown in Table 3.

### Table 4  Ranking of classifiers computed by the Friedman test

|     | SVM | GBDT | ANN | LRN | ROCKET | Random |
|-----|-----|------|-----|-----|--------|--------|
| Ranking | 2.00 | 3.93 | 4.75 | 1.11 | 3.05   | 6.00   |

![Fig. 12](image1.png) Performance of TP approaches in terms of TDFF across three datasets: Cisco, ABB, and Google

![Fig. 13](image2.png) Average error ranking of the evaluated classifiers in terms of TDFF. The leftmost performs the best

![Fig. 14](image3.png) Performance of TP approaches in terms of TDLF across five datasets: Cisco, ABB, Google, Aug-Cisco, and AugABB
In terms of the ML model training time (TRAIN), LRN performs the best, followed by the GBDT approach. SVM is the third best-performing approach in terms of training time, followed by ANN.

In terms of the total running time of the prioritization algorithm (PART), Random has the best performance. This is expected since it uses a basic random test selection which is computationally cheap. The results further show that all ML approaches outperform ROCKET, which is expected. The ANN approach has the best performance out of all ML approaches. GBDT is the next best-performing approach, followed by LRN and SVM. Average TRAIN and PART times for the three datasets Cisco, ABB, and Google are shown in Table 6.

5.4 Threats to validity

**Internal validity** refers to potential errors in our implementation of the ML models and their parameter tuning. To minimize this threat, we have carefully developed and inspected our algorithms. In addition, we have used a reliable development framework Keras for the implementation. Regarding parameters used in ML models, they have a great impact on model performance. Therefore, we have explored many different model configurations and selected the superior set of hyper parameters for the learned models.

**External validity** relates to the representativeness of the experimental subjects used in the study and the generalizability of our results. We used three datasets made available by three large companies, and we believe they are representative of real-world CI projects. While these datasets have a different number of CI cycles and a different number of test cases, which contributes to the generalizability of our results, there are other CI projects with even more different numbers of CI cycles and test set sizes. Therefore, we can generalize our results for CI projects with a similar number of test cases and the size of CI cycles, as well as the type of tests (integration tests). Furthermore, the three real-world datasets used in the study have a low percentage of failed test cases, as shown in Table 1, which is another threat to the generalizability of our results, because the ML models in general are sensitive to imbalanced data. To mitigate this threat, we generated two augmented datasets using a well known technique for dealing with the class imbalance problem SMOTE Lemaître et al. (2017), combining under and oversampling to create well-balanced synthetic data from the original test suites. The augmentation does not considerably affect the representativeness of real CI projects, as some CI projects may have a higher number of faults.

### Table 5 Ranking of classifiers computed by the Friedman test

|       | SVM | GBDT | ANN | LRN | ROCKET | Random |
|-------|-----|------|-----|-----|--------|--------|
| Ranking | 2.10 | 4.00 | 5.01 | 1.05 | 2.85   | 5.95   |
Construct validity means potential threats of using irrelevant metrics to evaluate and compare TP approaches. In this study, we aim to measure fault detection effectiveness of various TP approaches and their time-effectiveness, i.e., the capability of early fault detection and ranking time. To this end, we used APFD to measure fault-detection effectiveness and TDFT, TDLF, TRAIN, and PART metrics to measure time effectiveness. APFD is the most commonly used metric in TP research. TDFT, TDLF, TRAIN, and PART are intuitive metrics to measure time effectiveness of TP approaches.

Conclusion validity addresses potential incorrect conclusions made by the study, for example, regarding the effectiveness of the evaluated TP approaches. However, in this work, we compare different TP approaches one against the other, discussing their relative effectiveness on the selected datasets, and do not make any conclusions about the absolute effectiveness of the evaluated TP approaches.

### 6 Discussion and conclusion

Test prioritization in continuous integration has the potential to improve the effectiveness and speed of fault detection. Machine learning has recently been proposed as an efficient approach for improving the scalability of test prioritization. Motivated by these findings, we set out to understand the relative fault-prediction performance of selected ML approaches for test case prioritization in continuous integration. We specifically focus on two parameters of continuous integration: test history size used for training ML models for test prioritization and the size of the time budget available for CI cycles.

We selected four ML approaches that have shown good performance in test case prioritization in the literature (support vector machines, gradient boosting decision trees, neural networks, and LambdaRank with neural network) and designed a systematic experimental study comparing the four ML approaches one against the other and against the two heuristics for test prioritization. We compared the approaches in terms of time-effectiveness and fault-prediction effectiveness of prioritized test suites, answering three research questions. Next, we draw the main findings from the study.

|                | Cisco   | ABB     | Google  |
|----------------|---------|---------|---------|
|                | TRAIN [s] | PART [s] | TRAIN [s] | PART [s] | TRAIN [s] | PART [s] |
| **LRN**        | 25.00   | 2.00    | 60.00   | 3.00    | 155.00   | 17.00   |
| **SVM**        | 40.00   | 2.25    | 95.00   | 3.00    | 190.00   | 18.00   |
| **GBDT**       | 35.00   | 1.50    | 90.00   | 2.00    | 175.00   | 15.00   |
| **ANN**        | 50.00   | 1.35    | 105.00  | 1.90    | 199.00   | 10.00   |
| **ROCKET**     | -       | 65.00   | -       | 125.00  | -        | 3050.00 |
| **Random**     | -       | 1.25    | -       | 1.80    | -        | 9.00    |

LRN has the best performance among all ML-based approaches in terms of the time to detect the first fault (TDFF), time to detect the last fault (TDLF), and model training time, also outperforming heuristic-based approaches.
6.1 Key findings

**RQ1** investigates how does the length of test execution history used for learning to prioritize test cases impact the fault-prediction performance of ML approaches? We collected the test execution history of three industrial datasets, *Cisco, ABB, and Google*, and trained four ML classifiers on the collected test history to learn to detect failing test cases. During training, we varied the size of the test history from 20% of the available history with increments of 20%. For these datasets, our findings indicate that the optimal size of test history used for learning to prioritize is from 40 to 60%. For the *Cisco* and *ABB* datasets, the optimal size of test history was 60%, while it was 40% for the *Google* datasets. This may be due to the number of test cycles in each of these datasets: 110 for the *Cisco* dataset, 100 for the *ABB* dataset, and 2259 for the *Google* dataset. More test cycles mean longer test history, which may imply that a lower percentage of it should be used for learning to prioritize test cases. Finally, when comparing different ML models for test prioritization, we observed that the performance of ANN was the least sensitive to using older test history.

**RQ2** studies which ML approach is more effective in predicting test cases with higher fault-detection effectiveness, for a given time budget, and how do they compare to heuristic-based test case prioritization approaches? We compared the effectiveness of fault detection (APFD) of four ML approaches trained with the best configuration of test history one against another and against two heuristics when different time budget is available for testing. On these datasets, our results show that the best-performing approach for a short time budget in terms of the APFD metric is LRN. Its performance is comparable with ROCKET. Specifically, LRN performs better for longer time budgets compared to ROCKET, and ROCKET performs better for shorter time budgets compared to LRN for the *Cisco* and *ABB* datasets. The next best-performing approach is GBDT, followed by SVM. Our results indicate that ANN has the worst fault-detection performance for a short time budget compared to the other evaluated ML-based approaches. However, Random showed the absolute worst fault-detection performance out of all evaluated approaches.

**RQ3** evaluates which ML approach is more time-efficient, and how do they compare to heuristic-based test case prioritization approaches? We compare training time (TRAIN), ranking time (PART), and time to detect the first (TDFF) and last fault (TDLT) of four ML approaches and two heuristics for TP. Our results show that in terms of TDFF, the best-performing approach is LRN, which is comparable to SVM. The next best-performing approach is ROCKET, followed by GDBT. The ANN model has the worst ability to detect faults early compared to all evaluated ML-based approaches. Random has the worst performance out of all evaluated approaches. In terms of TDLF, the results show different performances of the TP approaches across non-augmented and augmented datasets. This is because the non-augmented datasets have a low percentage of failing test cases, and the faults can be detected sooner than for the augmented datasets with a higher percentage of failing test cases. Comparing the ML approaches, LRN shows to be a superior approach, followed by SVM and ROCKET which have comparable performance. The next best-performing approach is GBDT, followed by ANN. Random shows the worst performance. In terms of ML model training time (TRAIN), LRN performs the best, followed by the GBDT approach, further followed by the SVM approach, which is finally followed by the ANN approach. In terms of the total running time of the prioritization algorithm (PART), Random shows the best performance,
which is expected, since it uses a computationally cheap random test selection. Our results show that all ML approaches outperform ROCKET. The ANN approach has the best performance out of all ML approaches. GBDT is the next best-performing approach, followed by LRN and SVM.

6.2 Implications for practice and research

Implications for practice RQ1: The amount of test history used for test case prioritization should be carefully selected, as it impacts the performance of test case prioritization. Shorter history is more suited for projects with frequent CI cycles. RQ2: LRN showed to be the best-performing approach on the task of test case prioritization when different time budgets are available for testing, with its performance reaching saturation at about 95% of entire time budget. RQ3: LRN showed the best performance in terms of time to detect the first and last fault and model training time.

Implications for research RQ1: More experimental studies on the impact of test execution history on the effectiveness of test case prioritization for projects with a varied number of CI cycles are needed. RQ2: Further evaluation of LRN in comparison with more ML algorithms for different time budgets and using more evaluation datasets may discover interesting findings. RQ3: Further evaluation of ML-based approaches, including LRN, for test case prioritization to achieve an increased fault detection effectiveness of CI tests, while considering CI factors besides test history and time budget, is desired.

Acknowledgements This work is supported by the Research Council of Norway, grant no 287329.

Author contributions Dusica Marijan made the conceptual design, performed the experiments, and wrote the manuscript.

Data availability The datasets generated during and/or analyzed during the current study are available from the corresponding author upon request.

Declarations

Conflict of interest The author declares no competing interests.

References

Ali, S., Hafeez, Y., Hussain, S., & Yang, S. (2020). Enhanced regression testing technique for agile software development and continuous integration strategies. Software Quality Journal, 28(2), 397–423.

Bertolino, A., Guerriero, A., Miranda, B., Pietrantuono, R., & Russo, S. (2020, June). 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 (pp. 1–12). https://doi.org/10.1145/3377811.3380369

Busjaeger, B., & Xie, T. (2016). Learning for test prioritization: An industrial case study. In Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, Association for Computing Machinery, New York, NY, USA (pp. 975–980). https://doi.org/10.1145/2950290.2983954

Calvo, B., & Santafé Rodrigo, G. (2016). scmamp: Statistical comparison of multiple algorithms in multiple problems. The R Journal, 8, 248–256.
Chen, J., Lou, Y., Zhang, L., Zhou, J., Wang, X., Hao, D., & Zhang, L. (2018). 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, ESEC/FSE 2018, Association for Computing Machinery, New York, NY, USA (pp. 656–667). https://doi.org/10.1145/3236024.3236053

Elbaum, A. M. S., & Penix, J. (2014). The google dataset of testing results. Retrieved July 18, 2023, from https://code.google.com/p/google-shared-dataset-of-test-suite-results

Elbaum, S., Rothermel, G., & Penix, J. (2014). Techniques for improving regression testing in continuous integration development environments. In Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (pp. 235–245).

Grano, G., Titov, T. V., Panichella, S., & Gall, H. C. (2018). How high will it be? Using machine learning models to predict branch coverage in automated testing. In 2018 IEEE Workshop on Machine Learning Techniques for Software Quality Evaluation (MALETSQe) (pp. 19–24). https://doi.org/10.1109/MALTESQE.2018.8368454

Hasnain, M., Pasha, M. F., Lim, C. H., & Ghan, I. (2019). Recurrent neural network for web services performance forecasting, ranking and regression testing. In 2019 Asia-Pacific Signal and Information Processing Association Annual Summit and Conference (APSIPA ASC) (pp. 96–105). https://doi.org/10.1109/APSIPAASC47483.2019.9023052

Hemmatti, H., Fang, Z., Mantyla, M. V., & Adams, B. (2017). Prioritizing manual test cases in rapid release environments (p. 27). Verification and Reliability: Software Testing.

Jahan, H., Feng, Z., Mahmud, S., & Dong, P. (2019). Version specific test case prioritization approach based on artificial neural network. Journal of Intelligent and Fuzzy Systems, 36, 6181–6194.

Khaitibyarbini, M., Isa, M. A., Jawawi, D. N., & Tuneng, R. (2018). Test case prioritization approaches in regression testing: A systematic literature review. Information and Software Technology, 93, 74–93.

Lachmann, R., Schulze, S., Nieke, M., Seidl, C., & Schaefer, I. (2016). System-level test case prioritization using machine learning. In 15th International Conference on Machine Learning and Applications (pp. 361–368).

Lemaitre, G., Nogueira, F., & Aridas, C. K. (2017). Imbalanced-learn: A python toolbox to tackle the curse of imbalanced datasets in machine learning. Journal of Machine Learning Research, 18(17)-5. http://jmlr.org/papers/v18/16-365

Lima, J. A. P., & Vergilio, S. R. (2022). A multi-armed bandit approach for test case prioritization in continuous integration environments. IEEE Transactions on Software Engineering, 48(2), 453–465. https://doi.org/10.1109/TSE.2020.2992428

Machalica, M., Samylkin, A., Porth, M., & Chandra, S. (2019). Predictive test selection. In 2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP) (pp. 91–100). https://doi.org/10.1109/ICSE-SEIP2019.00018

Mahdieh, M., Mirian-Hosseinabadi, S. H., Etemadi, K., Nosrati, A., & Jalali, S. (2020). Incorporating fault-proneness estimations into coverage-based test case prioritization methods. Information and Software Technology, 121, 106269.

Marjani, D. (2015). Multi-perspective regression test prioritization for time-constrained environments. In 2015 IEEE International Conference on Software Quality, Reliability and Security (pp. 157–162). https://doi.org/10.1109/QRS.2015.31

Marjani, D., Gotlieb, A., & Sen, S. (2013). Test case prioritization for continuous regression testing: An industrial case study. In 2013 IEEE International Conference on Software Maintenance (pp. 540–543). https://doi.org/10.1109/ICSM.2013.91

Marjani, D., Gotlieb, A., & Liaaen, M. (2019). A learning algorithm for optimizing continuous integration development and testing practice. Software: Practice and Experience, 192–213. https://doi.org/10.1002/spe.2661

Marjani, D., & Liaaen, M. (2016). Effect of time window on the performance of continuous regression testing. In 2016 IEEE International Conference on Software Maintenance and Evolution (ICSME) (pp. 568–571). https://doi.org/10.1109/ICSME.2016.77

Marjani, D., & Liaaen, M. (2017). Test prioritization with optimally balanced configuration coverage. In 2017 IEEE 18th International Symposium on High Assurance Systems Engineering (HASE) (pp. 100–103). https://doi.org/10.1109/HASE.2017.26

Marjani, D., Liaaen, M., Gotlieb, A., Sen, S., & Ieva, C. (2017). Titan: Test suite optimization for highly configurable software. In 2017 IEEE International Conference on Software Testing, Verification and Validation (ICST) (pp. 524–531). https://doi.org/10.1109/ICST.2017.60

Marjani, D., & Liaaen, M. (2018). Practical selective regression testing with effective redundancy in interleaved tests. In Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice, ICSE-SEIP '18, Association for Computing Machinery, New York, NY, USA (pp. 153–162). https://doi.org/10.1145/3183519.3183532
Marijan, D., Liaaen, M., & Sen, S. (2018). DevOps improvements for reduced cycle times with integrated test optimizations for continuous integration. In 2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC) (Vol. 1, pp. 22–27). https://doi.org/10.1109/COMPSAC.2018.00012

Mirarab, S., & Tahvildari, L. (2008). An empirical study on bayesian network-based approach for test case prioritization. In 2008 1st International Conference on Software Testing, Verification, and Validation (pp. 278–287). https://doi.org/10.1109/ICST.2008.57

Niu, N., Brinkkemper, S., Franx, X., Partanen, J., & Savolainen, J. (2018). Requirements engineering and continuous deployment. IEEE Software, 35(2), 86–90.

Parnin, C., Helms, E., Atlee, C., Boughton, H., Ghattas, M., Glover, A., Holman, J., Micco, J., Murphy, B., Savor, T., et al. (2017). The top 10 adages in continuous deployment. IEEE Software, 34(3), 86–95.

Rosenbauer, L., Stein, A., Maier, R., Patzel, D., & Hahner, J. (2020). Xcs as a reinforcement learning approach to automatic test case prioritization. In Proceedings of the 2020 Genetic and Evolutionary Computation Conference Companion, 1798–1806.

Rothermel, G., Untch, R. H., Chu, C., & Harrold, M. J. (2001). Prioritizing test cases for regression testing. IEEE Transactions on Software Engineering, 27, 929–948.

Savor, T., Douglas, M., Gentili, M., Williams, L., Beck, K., & Stumm, M. (2016). Continuous deployment at facebook and oanda. In 2016 IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C) (pp. 21–30). IEEE.

Sharif, A., Marijan, D., & Liaaen, M. (2021). Deeporder: Deep learning for test case prioritization in continuous integration testing. In 2021 IEEE International Conference on Software Maintenance and Evolution (ICSME) (pp. 525–534). https://doi.org/10.1109/ICSME52107.2021.00053

Shi, A., Zhao, P., & Marinov, D. (2019). Understanding and improving regression test selection in continuous integration. In 2019 IEEE 30th International Symposium on Software Reliability Engineering (ISSRE) (pp. 228–238). IEEE.

Shi, T., Xiao, L., & Wu, K. (2020). Reinforcement learning based test case prioritization for enhancing the security of software. In 2020 IEEE 7th International Conference on Data Science and Advanced Analytics (DSAA) (pp. 663–672).

Srikanth, H., Cashman, M., & Cohen, M. B. (2016). Test case prioritization of build acceptance tests for an enterprise cloud application: An industrial case study. Journal of Systems and Software, 119, 122–135.

Publisher’s Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Springer Nature or its licensor (e.g. a society or other partner) holds exclusive rights to this article under a publishing agreement with the author(s) or other rightsholder(s); author self-archiving of the accepted manuscript version of this article is solely governed by the terms of such publishing agreement and applicable law.

Dusica Marijan is a Senior Research Scientist with the Department of Validation Intelligence for Autonomous Software Systems, Simula Research Laboratory, Oslo, Norway. Prior to Simula, she worked as a Senior Software Engineer in the consumer electronics industry. She has coauthored over 50 referred articles in the software engineering and artificial intelligence communities, and developed several software tools for testing critical software systems. Her research interests lie at the intersection of software engineering and machine learning for improving the trustworthiness (e.g., security, privacy, and robustness) of complex software systems.