Comparative Study of Machine Learning Test Case Prioritization for Continuous Integration Testing

Dusica Marijan

Simula Research Laboratory, Norway

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 length of test history used for training the classifiers. In evaluation, we use real-world industrial datasets from a continuous integration practice. The results show that different machine learning models have different performance

*Corresponding author
Email address: dusica@simula.no (Dusica Marijan)

for different size of test history used for model training and for different time budget 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

---

1. Introduction

Continuous integration (CI) is an agile software development practice where software is realeased 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. Software testing is an integral step running iteratively and successively as part of continuous code integration. Each code integration is followed by an integration testing iteration, which is typically extensive, to prevent breaking a build. CI testing requires short turnaround between starting test execution and detecting faulty regressions, 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 [1, 2, 3, 4, 5, 6]. As a response to this challenge, researchers have proposed various test selection, minimization, and prioritization [7, 8, 9, 10, 11, 12, 13] 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. [14] suggests that history-based TP is an effective approach for rapid release software, while [15] 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, [16] 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.

Now 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 Figure 1

Following a standard practice, code changes committed by developers are regression tested before they can be deployed to production. Several hundreds of changes made on a daily basis trigger the execution of several thousands of test cases. Change impact analysis is run to select the test cases impacted by the change. However, all impacted test cases cannot not fit the available time budget, and moreover, not all impacted test cases are equally useful in detecting faults. If test engineers were to manually select a subtest of tests produced by change impact analysis which they believe have the highest chance of detecting faults, such a process would be highly time-ineffeicient. 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 [17] computes an ordered set of test cases that are impacted by the code changes, and that are of the highest historical fault detection ability. With this approach, test engineers can detect up to 30% more regression faults compared to manual test selection

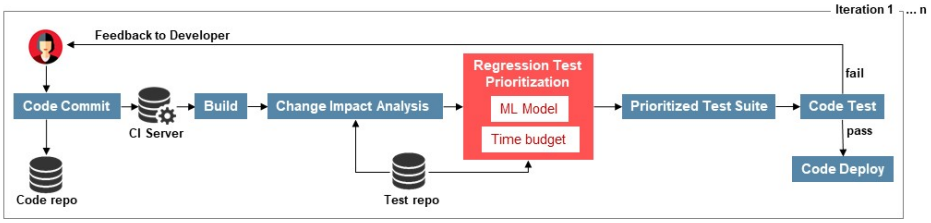

Figure 1: An iteration of CI testing consists of a code commit, build, and test phase, before code deployment. Change impact analysis and regression test prioritization are used to speed up testing. Regression test prioritization can use test history to learn to prioritize test cases and ML to scale test prioritization process to the size of large test history.

guided by tester's expertize, for the same time budget. 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 a 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 a 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 budget (budget for testing) and when we used different size 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.

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* [17] and *Random* test selection. We run the experiments on three industrial data

sets from the CI practice: *Cisco*, *ABB*, *Google*.

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.
- Systematic evaluation of the effectiveness of ML based test case prioritization approaches relative to variable time budget across CI cycles.
- Systematic comparison of the best-performing ML-based test case prioritization approaches reported in literature, evaluated on three industrial datasets.

The paper is structured as follows. In Section 2, we review related work. In Section 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 Section 5 presents the experimental results. We discuss the key findings of the study and conclude the paper in Section 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 and subset of test cases [18]. The approach was shown to reduces 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 [19]. It studies test case distribution analysis evaluating the fault detection capability of actual regression testing. The approach has been used in practice, and has shown to significantly reduce testing cost. 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) [20], to learn a binary classifier to order test cases based on historical information. The approach has shown to outperform non-ML-based test case prioritization approaches in terms of fault-detection effectiveness. Lachmann [21] 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 [22] 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 evaluation candidate in our study.

Several test case prioritization approaches have been proposed using different forms of neural netowrks, such as Bayesian network [23], NN [24], ANN [25], RNN [26]. Specifically, [23] integrates a feedback mechanism and a change information gathering strategy to estimate the probability of a test case to find bugs. The approach has showed to enable early fault detection. [24] prioritizes test cases using a NN approach and the fault-proneness distribution of different code areas. The approach has showed to improve the effectiveness of coverage based test case prioritization. [25] uses the combination of test case complexity information and software modification information to train an ANN, to enable early fault detection. The approach has showed to improve fault detection effectiveness. [26] 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 a 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 [27] or on using simpler ML models for RL policy design [28]. Lima proposes a multi-armed bandit (MAB) approach to test case prioritization in

continuous integration [29], 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 [30]. Furthermore, Bertolino [31] 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.

In the same study [31], 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 each CI cycle C_i, denoted as cycle history, where $i = 1...n$ and n is the number of CI cycles. 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 test suite T contains the pass/fail execution status and execution time of each test case t_i.

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 probability of detecting faults. In addition to fault detection effectiveness, some approaches use test execution time t as another prioritization criteria, which can be combined

together [17] 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 Figure 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_i belonging to a regression test suite $T = \{T_1, T_2, ..., T_n\}$, the goal of learning is to find a function $g : T \to C$, mapping the test case T_i 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 $C \in \{0, 1\}$. 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, and n denotes the number of CI cycles (test executions). 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 [17].

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 [30], and have found the total runtime of the

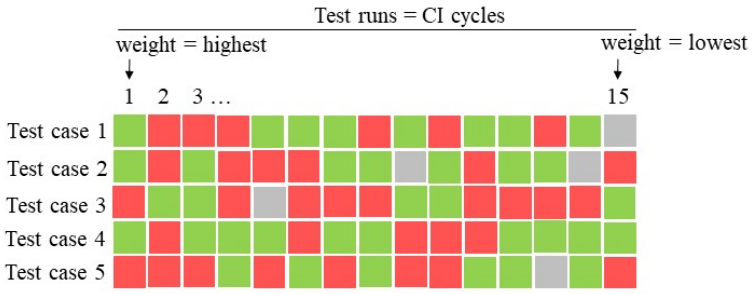

Figure 2: Test history consisting of 15 CI cycles. Cycle 1 is the most recent and has the highest weight, while cycle 15 is the oldest and has the lowest wight. Test cases can change execution result between pass and fail in consecutive executions (CI cycles). Red: fail, Green: pass, Grey: inconclusive. In this work, we only deal with pass and fail test results.

RL approach to be around 50 times higher than the runtime of the NN approach. This is consistent with the results reported by Bertolino [31]. 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. For our evaluation study we selected the following ML classifiers: Support Vector Machine (SVM) classifier, Artificial Neural Network (ANN) classifier, Gradient Boosted Decision Tree (GBDT) classifier, and LambdaRank with NN (LRN) classifier.

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 Section 3 with the aim of answering the following research questions:

RQ1 How does the length of test execution history used for learning to prioritize test cases impact the fault-prediction performance of ML approaches?

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?

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 system-level testing in CI: *Cisco*, *ABB*, *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 [32]. The datasets contain the 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.

We summarize the datasets in Table 1.

Table 1: Evaluation datasets.

Dataset	**# test cases**	**# test executions**	**% failed test cases**
Cisco	550	6050	0.43
ABB	1488	149700	0.28
Google	5507	12439910	0.01

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* [17] that has previously shown to improve the effectiveness of manual practice of test selection at Cisco, and the *Random* approach.

[1] https://bitbucket.org/HelgeS/atcs-data/src/master/

[2] https://code.google.com/archive/p/google-shared-dataset-of-test-suite-results/

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 such to maximize early fault detection. More information about *ROCKET* can be found here [17]. We varied the length of 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 in terms of the following metrics:

APFD as the weighted average of the percentage of faults detected.

TDFT as the time to detect the first fault by a prioritized test suite.

TDLF as the time to detect the last fault by a prioritized test suite.

TRAIN as the training time of a ML model for test prioritization.

PART as the 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 $H1$-$H5$. 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 [33], 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 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, 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.

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 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, 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 the minimal MSE and SD.

5. Results and Analysis

In this section, we first analyse the experimental results answering the research questions, and discuss 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 Figure 3. 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 a less negative effect on ANN compared to other ML approaches approaches. However, the results show that $ROCKET$ is more negatively affected by using older test history than the ML approaches. 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 the 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 $H3$ on the *Cisco* and ABB datasets.

In summary, the results indicate that the size of test execution history used for learning to prioritize affects the fault-prediction performance of TP approaches. For the *Cisco* and ABB experimental datasets, the optimal size of test history has shown to be $H3$, which corresponds to 60% of test history. For the *Google* datasets, the optimal size of test history 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* datasets 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 shorter history.

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, $H3$ for the *Cisco* and ABB datasets and $H2$ 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 Figure 4. Columns $B1$-$B5$ correspond to five different values of the time budget, starting from 20% of the average overall time required to run a test suite, with increments of 20%.

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 longer time budget enables higher fault-detection, as there

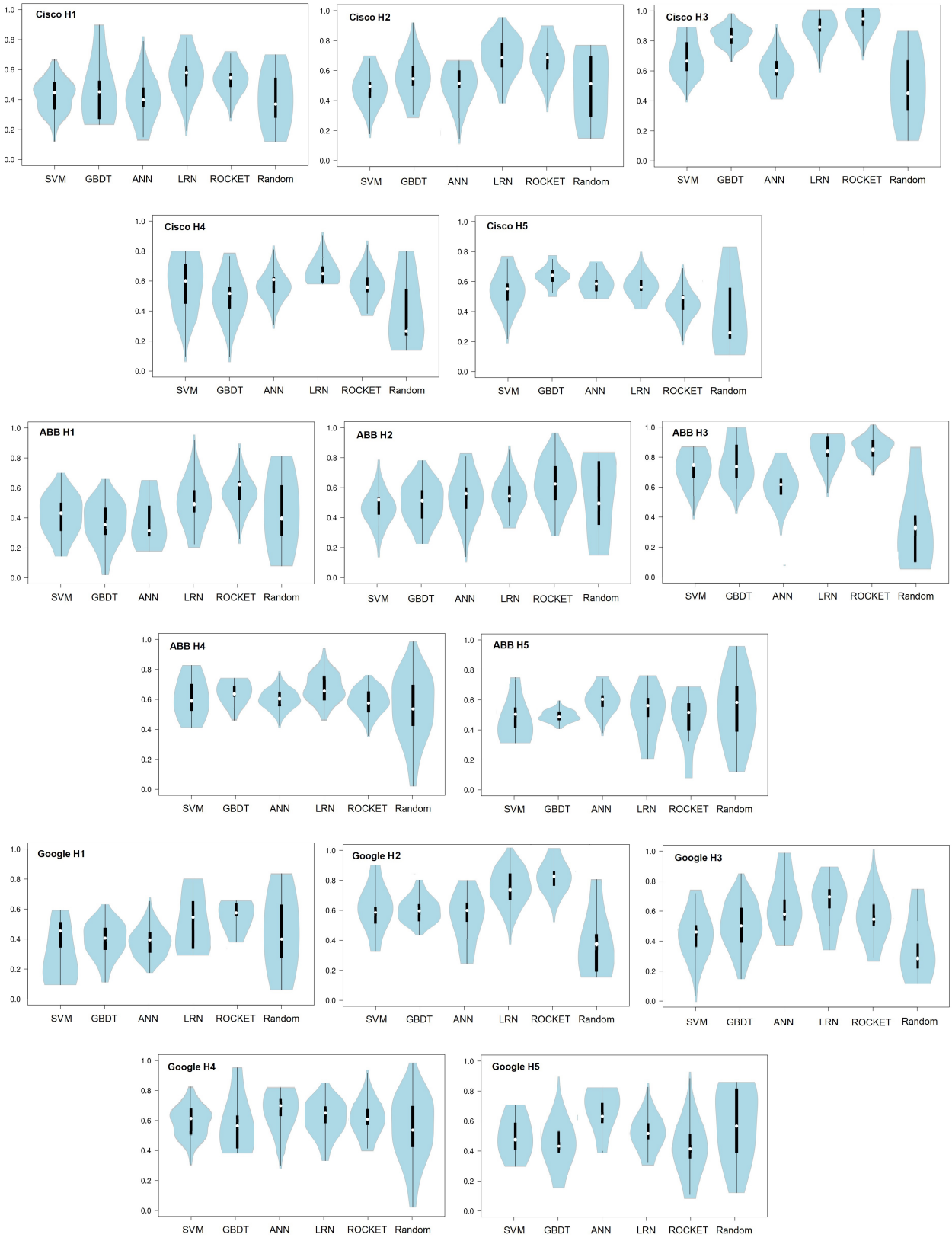

Figure 3: Performance of TP approaches in terms of APFD for different size of test history used for learning to prioritize ($H1$-$H5$) across three datasets: *Cisco*, *ABB*, and *Google*.

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 it fault-detection effectiveness improves for larger time budgets.

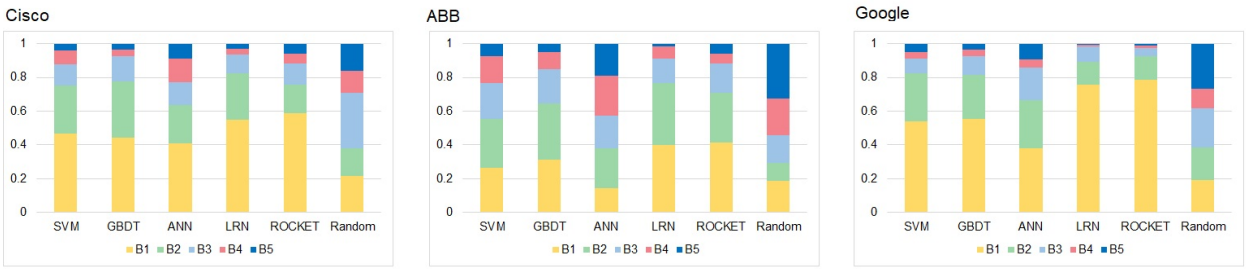

Figure 4: Performance of TP approaches in terms of average APFD for different size of test budget ($B1 - B5$) across three datasets: *Cisco*, *ABB*, *Google*.

5.3. RQ3: Time Effectiveness

Time effectiveness is measured using four metrics: TDFF, TDLF, TRAIN, and PART. 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.

In terms of the time to detect the last fault (TDLF), 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 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 2.

Table 2: Time metrics: average TRAIN and PART across *Cisco*, *ABB*, and *Google* datasets.

	Cisco		ABB		Google	
	TRAIN [s]	PART [s]	TRAIN [s]	PART [s]	TRAIN [s]	PART [s]
LRN	25	2	60	3	155	17
SVM	40	2.25	95	3	190	18
GBDT	35	1.5	90	2	175	15
ANN	50	1.35	105	1.9	199	10
ROCKET	-	65	-	125	-	3050
Random	-	1.25	-	1.8	-	9

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 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. Our results show that the length of test execution history used for learning to prioritize test cases impacts the fault-prediction performance of ML approaches. For these datasets, our findings indicate that the optimal size of test history used for learning to prioritize is from 40% to 60%. When comparing different ML models for test prioritization, we observed that the performance of ANN was the least sensitive to using older test history. At the same time, the ANN approach showed the worst performance for short test history among all other evaluated ML approaches. Next, our results show that in terms of fault-prediction effectiveness for a given time budget, the best performing approach for a short time budget in terms of the APFD metric is LRN, while ANN showed to have the worst fault-detection performance for a short time budget compared to the other evaluated ML-based approaches. Finally, in terms of time-effectiveness (time to detect the first and the last fault), the best performing approach is LRN. In terms of ranking time, the best performing ML approach is ANN.

References

References

[1] N. Niu, S. Brinkkemper, X. Franch, J. Partanen, J. Savolainen, Requirements engineering and continuous deployment, IEEE software 35 (2) (2018) 86–90.

[2] T. Savor, M. Douglas, M. Gentili, L. Williams, K. Beck, M. Stumm, Continuous deployment at facebook and oanda, in: 2016 IEEE/ACM 38th

International Conference on Software Engineering Companion (ICSE-C), IEEE, 2016, pp. 21–30.

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

[4] D. Marijan, A. Gotlieb, M. Liaaen, A learning algorithm for optimizing continuous integration development and testing practice, in: Software: Practice and Experience, 2019, pp. 192–213. doi:doi.org/10.1002/spe.2661.

[5] D. Marijan, M. Liaaen, 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, 2018, p. 153–162. doi:10.1145/3183519.3183532.
URL https://doi.org/10.1145/3183519.3183532

[6] D. Marijan, M. Liaaen, S. Sen, 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. 01, 2018, pp. 22–27. doi:10.1109/COMPSAC.2018.00012.

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

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

[9] G. Rothermel, R. H. Untch, C. Chu, M. J. Harrold, Prioritizing test cases

for regression testing, IEEE Transactions on software engineering 27 (2001) 929–948.

[10] D. Marijan, Multi-perspective regression test prioritization for time-constrained environments, in: 2015 IEEE International Conference on Software Quality, Reliability and Security, 2015, pp. 157–162. doi:10.1109/QRS.2015.31.

[11] D. Marijan, M. Liaaen, Test prioritization with optimally balanced configuration coverage, in: 2017 IEEE 18th International Symposium on High Assurance Systems Engineering (HASE), 2017, pp. 100–103. doi:10.1109/HASE.2017.26.

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

[13] S. Sen, D. Marijan, C. Ieva, A. Grime, A. Sander, Modeling and verifying combinatorial interactions to test data intensive systems: Experience at the norwegian customs directorate, IEEE Transactions on Reliability 66 (1) (2017) 3–16. doi:10.1109/TR.2016.2618121.

[14] H. Hemmati, Z. Fang, M. V. Mantyla, B. Adams, Prioritizing manual test cases in rapid release environments, Software Testing, Verification and Reliability 27.

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

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

[17] D. Marijan, A. Gotlieb, S. Sen, Test case prioritization for continuous regression testing: An industrial case study, in: 2013 IEEE International Conference on Software Maintenance, 2013, pp. 540–543. doi:10.1109/ICSM.2013.91.

[18] M. Machalica, A. Samylkin, M. Porth, S. Chandra, Predictive test selection, in: 2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP), 2019, pp. 91–100. doi:10.1109/ICSE-SEIP.2019.00018.

[19] J. Chen, Y. Lou, L. Zhang, J. Zhou, X. Wang, D. Hao, 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, ESEC/FSE 2018, Association for Computing Machinery, New York, NY, USA, 2018, p. 656–667. doi:10.1145/3236024.3236053.
URL https://doi.org/10.1145/3236024.3236053

[20] B. Busjaeger, 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, FSE 2016, Association for Computing Machinery, New York, NY, USA, 2016, p. 975–980. doi:10.1145/2950290.2983954.
URL https://doi.org/10.1145/2950290.2983954

[21] R. Lachmann, S. Schulze, M. Nieke, C. Seidl, , I. Schaefer, System-level test case prioritization using machine learning, in: 15th International Conference on Machine Learning and Applications, 2016, p. 361–368.

[22] G. Grano, T. V. Titov, S. Panichella, H. C. Gall, 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 (MaLTeSQuE), 2018, pp. 19–24. doi:10.1109/MALTESQUE.2018.8368454.

[23] S. Mirarab, L. Tahvildari, An empirical study on bayesian network-based approach for test case prioritization, in: 2008 1st International Conference on Software Testing, Verification, and Validation, 2008, pp. 278–287. doi:10.1109/ICST.2008.57.

[24] M. Mahdieh, S.-H. Mirian-Hosseinabadi, K. Etemadi, A. Nosrati, S. Jalali, Incorporating fault-proneness estimations into coverage-based test case prioritization methods, Inf. Softw. Technol. 121 (2020) 106269.

[25] H. Jahan, Z. Feng, S. Mahmud, P. Dong, Version specific test case prioritization approach based on artificial neural network, in: Journal of Intelligent and Fuzzy Systems, Vol. 36, 2019, p. 6181–6194.

[26] M. Hasnain, M. F. Pasha, C. H. Lim, I. Ghan, 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), 2019, pp. 96–105. doi:10.1109/APSIPAASC47483.2019.9023052.

[27] T. Shi, L. Xiao, K. Wu, 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) (2020) 663–672.

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

[29] J. A. do Prado Lima, S. R. Vergilio, A multi-armed bandit approach for test case prioritization in continuous integration environments, IEEE Transactions on Software Engineering.

[30] A. Sharif, D. Marijan, M. Liaaen, Deeporder: Deep learning for test case prioritization in continuous integration testing, in: 2021 IEEE International

Conference on Software Maintenance and Evolution (ICSME), 2021, pp. 525–534. doi:10.1109/ICSME52107.2021.00053.

[31] A. Bertolino, A. Guerriero, B. Miranda, R. Pietrantuono, S. Russo, Learning-to-rank vs ranking-to-learn: Strategies for regression testing in continuous integration, in: In 42nd International Conference on Software Engineering (ICSE), 2020.

[32] A. M. S. Elbaum, J. Penix, The google dataset of testing results, Available: https://code.google.com/p/ google-shared-dataset-of-test-suite-results.

[33] D. Marijan, M. Liaaen, Effect of time window on the performance of continuous regression testing, in: 2016 IEEE International Conference on Software Maintenance and Evolution (ICSME), 2016, pp. 568–571. doi:10.1109/ICSME.2016.77.