Learning-to-Rank *vs* Ranking-to-Learn: Strategies for Regression Testing in Continuous Integration

Antonia Bertolino
antonia.bertolino@isti.cnr.it
ISTI - CNR
Pisa, Italy

Antonio Guerriero
antonio.guerriero@unina.it
Università di Napoli Federico II
Napoli, Italy

Breno Miranda
bafm@cin.ufpe.br
Federal University of Pernambuco
Recife, Brazil

Roberto Pietrantuono
roberto.pietrantuono@unina.it
Università di Napoli Federico II
Napoli, Italy

Stefano Russo
stefano.russo@unina.it
Università di Napoli Federico II
Napoli, Italy

ABSTRACT

In Continuous Integration (CI), regression testing is constrained by the time between commits. This demands for careful selection and/or prioritization of test cases within test suites too large to be run entirely. To this aim, some Machine Learning (ML) techniques have been proposed, as an alternative to deterministic approaches. Two broad strategies for ML-based prioritization are *learning-to-rank* and what we call *ranking-to-learn* (i.e., reinforcement learning). Various ML algorithms can be applied in each strategy. In this paper we introduce ten of such algorithms for adoption in CI practices, and perform a comprehensive study comparing them against each other using subjects from the Apache Commons project. We analyze the influence of several features of the code under test and of the test process. The results allow to draw criteria to support testers in selecting and tuning the technique that best fits their context.

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**.

KEYWORDS

regression testing, test selection, test prioritization, continuous integration, machine learning

ACM Reference Format:
Antonia Bertolino, Antonio Guerriero, Breno Miranda, Roberto Pietrantuono, and Stefano Russo. 2020. Learning-to-Rank *vs* Ranking-to-Learn: Strategies for Regression Testing in Continuous Integration. In *42nd International Conference on Software Engineering (ICSE '20), May 23–29, 2020, Seoul, Republic of Korea.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3377811.3380369

ICSE '20, May 23–29, 2020, Seoul, Republic of Korea

ACM ISBN 978-1-4503-7121-6/20/05…$15.00
https://doi.org/10.1145/3377811.3380369

1 INTRODUCTION

Continuous Integration (CI) is widely practiced in the software industry [21, 23] for its benefits in terms of release time and productivity [29]. Due to the frequent commits to the shared codebase, the cost of continuously performing regression testing escalates [22].

Regression testing has been investigated for decades [36]. However, most techniques for reducing its cost in traditional development cannot be applied at the scale of modern CI practices [5]. Scalability issues are due not only to the size of codebases and test suites, but also to the dynamicity of such environments [22]. To address the needs of regression testing in the context of CI, researchers actively chase lightweight and effective test selection and prioritization (TS&P) techniques.

An ideal TS&P technique for CI should be able to quickly identify a relevant subset of test cases that can *safely* and *timely* detect any potential regression introduced by the latest committed changes. Solutions are investigated along two main directions: heuristics to trade off precision and effort [5, 7, 34], and fully automated approaches leveraging Machine Learning (ML) algorithms [1, 28]. This paper features a TS&P approach that first picks a subset of test cases using a coarse-grained static selection approach as the one proposed in [34], and then prioritizes them through ML.

Prioritization is basically a *ranking* problem and naturally lends itself to be formulated as a learning task. In fact, several authors have investigated the use of ML for *prioritization* in CI, as we discuss in the next section. However, we still lack criteria for choosing the most appropriate technique to be applied in a certain situation. In this paper, we address the gap by experimentally evaluating ten ML algorithms which may be adopted for TS&P in CI, and analyzing the influence of features of the code under test (CUT) and of the test process. Nine of the ten ML algorithms have never been used before for test prioritization.

We consider two different learning strategies. *Learning-to-rank* (LTR) encompasses mainly supervised algorithms, proved useful in information retrieval and natural language processing [18]. In software engineering, LTR was successfully applied to defect prediction, to rank modules based on their defectiveness [35]. In test prioritization, LTR can be used to rank *test targets* (e.g., test cases or test classes) based on a *testing objective* (e.g., the chance of exposing failures). Being usually formulated as a supervised learning problem, LTR requires prior training. When the operating context differs

from the training one, the model may no longer be representative and may loose its prediction ability. This indeed may happen in CI.

An alternative strategy, suited in dynamic contexts, is *reinforcement learning* (RL). RL foresees an artificial *agent* that learns from the *environment* by observing its *state* and selects a proper action, either from a learned *policy* or by random exploration of possible actions. As a result of the action, the agent receives feedback in form of *rewards*; the goal is to take actions that maximize the reward. The agent is usually implemented with (shallow or deep) neural networks, mapping state-action pairs to rewards. We denote RL algorithms applied to ranking as *ranking-to-learn*, as opposed to *learning-ro-rank*, since they leverage the ranking at each step to improve the model's predictive ability. We foresee a potential benefit in using RTL for test prioritization in CI, due to its natural ability to adapt to changes in the test suites – with removed and newly added tests at every CI cycle - and to changes in the CI process.

In summary, the original contributions of this paper are:

- We present ways to formulate TS&P as a ML problem;
- We conduct the first experimental study comparing performance of LTR and RTL test prioritization algorithms in CI;
- We identify and study features of the CUT and of the CI process, which may affect the effectiveness of algorithms;
- We draw criteria for applying ML techniques to the TS&P problem in CI environments, thus supporting testers in selecting and tuning the strategy that best fits their context.

The paper is organized as follows. Section 2 surveys related work. Section 3 presents our TS&P techniques. Section 4 describes the experimental evaluation, and Section 5 presents the results. Threats to validity and guidelines are discussed in Section 6. Section 7 provides concluding remarks.

2 RELATED WORK

Research on TS&P in CI environments is today very active. A recent study analyzes change commits of almost one thousand Github projects, to understand the needs of regression test selection [34]. Various techniques have been proposed to efficiently identify those test cases that exercise the changed code [7, 13]. Machine Learning has been proposed for this purpose, too: Pang *et al.* [25] demonstrate that simple unsupervised learning algorithms such as k-means, based on coverage information, can be used with good results. However, at large scale it may be infeasible to collect coverage information [5]: in contrast, static approaches use program analysis to identify the code parts potentially affected by a change. Legunsen *et al.* [16] observed that class-level techniques may be as effective as dynamic ones. Following their conclusions, this work adopts a static approach to test selection at class-level (see Section 3).

Test suite prioritization techniques are reviewed by Khatibsyarbini *et al.* [12], who classify 69 primary studies between 1999 and 2016. They find that the three most used prioritization techniques are search-based (25%), coverage-based (18%), and fault-based (10%). The survey includes the work by Thomas *et al.* [30] using topic modeling (a text mining method) for similarity-based test prioritization, yet it does not identify an explicit class of ML prioritization techniques. Indeed, those more closely related to our work – i.e., [1, 28], discussed below - appeared only later.[1] The survey missed though a 2006 work by Tonella *et al.* [32], who apply the Case-Based Ranking algorithm to rank tests according to coverage, complexity metrics and historical data; when the ordering between two test cases cannot be decided, the user is asked to manually rank them.

Targeting CI environments, we opt for the analysis of fully automated ML techniques. These draw increasing interest today in many domains [26], including several software engineering tasks, as early surveyed by Zhang and Tsai in 2003 [37]. Use of ML in software testing is the focus of a recent mapping study by Durelli *et al.* [4], who noted a surge of research in this topic in the very last years. They identify four primary studies that use ML for test prioritization: in addition to the one by Tonella *et al.* already discussed [32], they are [1, 17, 28].

Lenz *et al.* [17] propose a ML strategy, which leverages test-related information to support various tasks, including test prioritization: they first use clustering of data derived by executing some example test cases to obtain groups of functionally related test cases; these clusters are then used to train a ML classifier.

Busjaeger and Xie [1] claim to be the first to reduce the problem of test prioritization to that of LTR: their model learns from a training set made up by past changes and by the tests observed for each of them; tests are binary labelled (pass/fail). A feature vector is created for each change/tests pair. Their model, experimented on a real-world dataset, includes five features: coverage data, test-file path similarity and test content similarity, failure history, and test age. The approach, based on a listwise LTR algorithm (see Section 3.2.1), showed significantly better results than existing heuristics relying on single features. We experiment a different listwise LTR algorithm, never used before, and compare it to other strategies.

Lachman *et al.* [15] apply the LTR SVM-Rank algorithm to black-box prioritization starting from test cases and failure reports in natural language (NL). They derive a dictionary of terms from the test cases and collect further metadata based on history and requirements. The evaluation shows that ML prioritization outperforms previously available manual approaches by experts. As we target CI environments, we do not consider tests and requirements in NL.

Spieker *et al.* [28] observe that existing prioritization techniques using historical information cannot properly account for changes in the testing context: on the one side, test cases can be removed from or added to the test suite; on the other, the testing focus could vary based on external factors. Therefore they propose RETECS, the first TS&P approach using RL with a shallow neural network agent. As a lightweight approach, it uses only failure history information. The evaluation on a real-world data set shows that performance comparable to deterministic approaches can be achieved after a learning stage of about 60 CI cycles, without training. We perform a more comprehensive analysis of RL algorithms, including multi-layer perceptron and random forest, besides a shallow network.

Finally, Elbaum *et al.* [5] propose a history-based TS&P strategy for CI, customized to the pre- and post-submit stages of the Google CI process, whereas our study does not assume any specific process.

3 APPROACH

In CI practices, testing is a time-constrained problem, typically dealt with by proper test selection and/or prioritization algorithms. The goal of the former is to select only those tests exercising the

[1] Reference [1] is dated November 2016, probably too late to appear in [12].

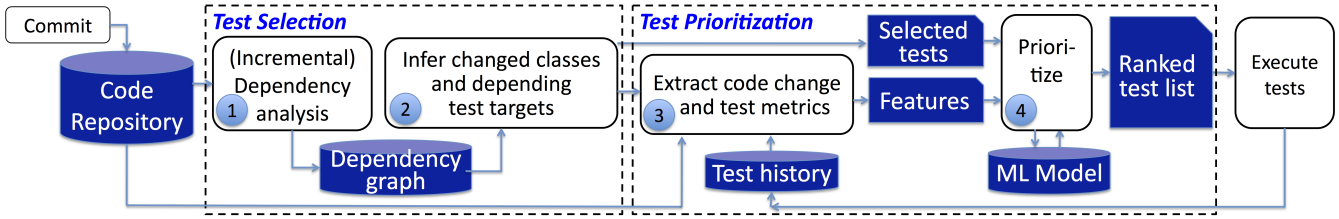


Figure 1: Overview of the test selection and prioritization process

code directly or indirectly affected by changes. The latter reorders the entire test suite so that tests with higher priority are run first. Selection alone may be insufficient if time between commits is so short that not all selected tests can be run, or when the goal is to avoid running many non-failing tests. Prioritization without selection may be unsatisfactory too, as it might act on non-relevant tests. Clearly, they can be combined, by selecting a subset of tests, then prioritizing them [22], or by prioritizing the suite, then selecting tests, e.g., by a temporal threshold excluding the lowest priority ones [28]. A process made up by a lightweight coarse-grained selection followed by ML-based prioritization is proposed here (Fig. 1).

3.1 Test selection

We adopt a conservative criterion for test selection, based on static class-level dependency analysis. Static techniques are preferred over dynamic ones since the latter are often impractical in CI environments due to runtime overhead. Lightweight dynamic techniques, like in [7], could be applied, but their reliance on runtime collection of dependencies is a hurdle in CI contexts, as it is time-consuming and, for programs with non-determinism, the result may be unsafe since collected dependencies may not cover all possible traces [16]. Class-level is preferred to method-level selection, as it is faster and cheaper, and even safer [16][34]. We exclude coverage-based test selection (and prioritization as well), which in CI may be expensive (due to the costs of instrumentation, recording and maintaining coverage data per release) and inaccurate (the quick cycles and code changes make coverage data imprecise and obsolete) [5][8][9][34].

In the first CI cycle, we build the class-level dependency graph;[2] at next commits, we consider the changed classes and update it accordingly (Step 1 in Fig. 1). By querying the graph, we obtain (Step 2) all the classes that transitively depend on the changed classes, along with the associated *test classes*, which are our *test targets*. These are then prioritized.

3.2 Test prioritization

The test targets prioritization criteria are *fault detection* and *execution time*. Several ways of combining these two criteria can be envisaged, depending on the testing objective. As usual in test prioritization, fault detection is assumed here as the primary criterion: the optimal ranking list is the one that orders all failing test targets first and non-failing targets after; within each of these two sublists, targets with shorter execution time are ranked first.

The features used to predict the ranking (derived in Step 3) are listed in Table 1.[2] The first three rows are code metrics of the class(es) under test. The last row refers to test metrics: test execution time and failure history of the test target up to the previous four commits.[3] Specifically, we compute the difference between the metrics' values of two consecutive commits. This study analyzes the following LTR and RTL strategies (for adoption in Step 4).

Table 1: Code and test metrics

Type	Metrics	Description
Program size	*AvgLine, AvgLineBlank, AvgLineCode, AvgLineComment, CountDeclFunction, CountLine, CountLineBlank, CountLineCode, CountLineCodeDecl, CountLineCodeExe, CountLineComment, CountSemicolon, CountStmt, CountStmtDecl, CountStmtExe, RatioCommentToCode*	Metrics related to the amount of lines of code, declarations, statements, and files
McCabe's cyclomatic complexity	*AvgCyclomatic, AvgCyclomaticModified, AvgCyclomaticStrict, AvgEssential, MaxCyclomatic, MaxCyclomaticModified, MaxCyclomaticStrict, MaxEssential, MaxNesting, SumCyclomatic, SumCyclomaticModified, SumCyclomaticStrict, SumEssential*	Metrics related to the control flow graph of functions and methods
Object-oriented metrics	*CountDeclClass, CountDeclClassMethod, CountDeclClassVariable, CountDeclExecutableUnit, CountDeclInstanceMethod, CountDeclInstanceVariable, CountDeclMethod, CountDeclMethodDefault, CountDeclMethodPrivate, CountDeclMethodProtected, CountDeclMethodPublic*	Metrics based on object-oriented constructs
Test history	*Number of failed tests in the current commit, Number of failed tests per test class n commits before the current one (n=1 to 4), Total execution time of all the tests of the test class, Last time the test class was run*	Metrics based on the history of tests execution

3.2.1 ***LTR.*** Training is done on a number of observations W, depending on the amount of history available; the resulting model is used to prioritize tests for next commits. The model needs to be re-trained from time to time: this is preferred to online learning when training is expensive (e.g., for a large codebase). The LTR strategy can use *pointwise*, *pairwise* or *listwise* algorithms [18].

Pointwise LTR. The ranking problem is transformed into classification, regression, or ordinal classification, and solved with respective existing methods. The training data are typically supervised learning data; given a sample (a test target), the algorithm predicts the class label, real number or grade label for the three cases. For example, in classification problems the score can be the probability of a test belonging to a class (e.g., high-priority and low-priority, in a binary formulation); in regression and ordinal classification problems, a function of the testing objective yielding a real priority number or a grade. The loss function in learning is pointwise in the sense that it is defined on a single object (feature vector).

Pairwise LTR. Ranking is transformed into a classification or regression problem, where a sample is a pair of test targets: a model can tell which test target has higher score than the other in a pair.

[2] We use SCITools Understand 2.0 (*http://scitools.com*).

[3] A too long history may not make sense in CI, as code changes and test outcomes generally change over time. The choice of 4 previous commits was first made in [28].

The goal is to minimize the average number of inversions in ranking, due to unordered pairs.

Listwise LTR. The problem is addressed in an intuitive way, as ranking lists are taken directly as samples in both learning and prediction. The approach trains a model able to assign scores to feature vectors and rank them accordingly. The goal is to minimize the difference between the predicted and the actual ranking lists.

In classification-based pointwise LTR, we consider four classes derived from the two above-mentioned prioritization criteria (fault detection and execution time), which are (in decreasing priority):
Class 3: *At least one failure* is detected running the test target, and the *execution time of the test target* is *shorter* than a threshold (computed as the median of execution times on the last W samples);
Class 2: *At least one failure* is detected running the test target, and its execution time is *longer* than the threshold;
Class 1: *No failure* is detected running the test target, and its execution time is *shorter* than the threshold;
Class 0: *No failure* is detected running the test target, and the *execution time of the test target* is *longer* than the threshold.

In regression problems, the relevance of test targets needs to be defined for the LTR algorithm to assign a score R_i to the i-th target. With the objective of prioritizing the failing tests first and, then, the shortest ones, the relevance function is defined as:

$$R_i = F_i(1 + e^{-T_i}) + (1 - F_i)e^{-T_i} = F_i + e^{-T_i} \quad (1)$$

where $F_i = 1$ if the test target fails at least once, 0 otherwise, and T_i is the execution time of the target. This ensures that a higher score is given to failing and shorter test targets compared to the others (i.e., failing/longer, non-failing/shorter and non-failing/longer having progressively lower scores).

3.2.2 ***RTL.*** Training is done *online*, namely when the agent updates its knowledge about state-action-reward. The reward is implemented as in the classification-based pointwise approach, i.e., with four classes. Here, the states are the test targets to prioritize. The policy is a function from states to actions, which initially is a loose approximation of the optimal policy, and is then refined over time by the gained experience. An action is the assignment of a priority to a test target by using the policy, preferring the actions rewarding more. Several approximators of the policy can be used; neural networks are the commonly used ones [33].

In our formulation, the approximator receives a state as input, and outputs a probability vector of class membership (for the four classes). The probability is used as priority score to rank tests attributed to the same class. To train the agent, a mechanism called *experience replay* is exploited [19]: the past experience of the agent (in terms of state, action, reward, next state) is stored separately in order to have the opportunity to reprocess it later and use for learning in different ways. In an online setting as ours, the *replay memory* keeps the experience information of the last N time steps, with N constrained by the limited memory capacity. When the capacity is reached, oldest experiences get replaced first. Training exploits a batch of experience, which can be sampled in many ways: we sample it randomly, giving the newer observations higher probability of selection than the older ones; given the i-th observation (the bigger the older), the selection probability is: $p_i = \frac{1/i}{\sum_{j=1}^{N} 1/j}$.

Table 2: Algorithms

Strategy	Class	Algorithm
LTR	Pointwise	K-Nearest Neighbor (KNN)
LTR	Pointwise	Random Forest (RF)
LTR	Pairwise	LambdaMART (L-MART)
LTR	Pairwise	MART
LTR	Pairwise	RankBoost
LTR	Pairwise	RankNet
LTR	Listwise	Coordinate ASCENT (CA)
RTL	Reinforcement Learning	Shallow Network (RL)
RTL	Reinforcement Learning	Multilayer Perceptron (RL-MLP)
RTL	Reinforcement Learning	Random Forest (RL-RF)

We consider three RTL variants, where the agent is: *i)* a *shallow network*, as the one used in the Spieker's model [28], but preceded by the test selection described in Section 3.1 and with a different reward (classification-based); *ii)* a *multilayer perceptron*; *iii)* a *classification* algorithm (namely, Random Forest).

4 EVALUATION

4.1 Algorithms

This work evaluates the ten algorithms listed in Table 2. They can be further classified into *ensemble* and *non-ensemble* algorithms: the former category includes: RF, RL-RF, RankBoost, MART, L-MART; the others are non-ensemble. The Weka[4] and Knime[5] tools were used for pointwise algorithms, and the RankLib library[6] for pairwise and listwise algorithms. The number of samples used for training, initially set to 2,000, is subject to sensitivity analysis. For supporting the independent verification and replication, we make available the Python implementation for the RTL strategy, the code for test selection (for Java and the Maven build system), the algorithms' settings as well as additional results not included here for the sake of space.[7]

4.2 Experimental factors

We investigate what factors make some ML algorithms behave better than others for test prioritization in a CI context. We focus on characteristics of the **code under test** and of the **CI process**. As for the former, we consider: *i)* the *variability* of the code/test metrics; *ii)* the *failure proneness* of the code, which causes more or less balanced datasets; *iii)* the code/test *metrics* that can be used as *features* for training and improving prediction. As for the latter, we consider: *i)* the *inter-commit time*, which determines the time available for performing TS&P; *ii)* the *cycle size* (i.e., size of the sample, or number of tests, per commit), which affects the *length of the history* available for learning.

4.3 Research Questions

The study addresses the following research questions:

[4] https://www.cs.waikato.ac.nz/ml/weka/.
[5] https://www.knime.com/knime-software.
[6] https://sourceforge.net/p/lemur/wiki/RankLib/.
[7] https://github.com/icse20/RT-CI

- **RQ1**. How do the selected algorithms perform in a CI context in terms of prioritization effectiveness and cost?
  - **RQ1.1** Which algorithm performs better?
  - **RQ1.2** Which of the four strategies (*RTL; pointwise, pairwise and listwise LTR*) performs better?
  - **RQ1.3** Which category performs better between *non-ensemble* and *ensemble* algorithms?
- **RQ2**: What is the influence of code characteristics?
  - **RQ2.1**: What is the influence of CUT variability?
  - **RQ2.2**: What is the influence of the code failure proneness?
  - **RQ2.3**: How many and which features are important?
- **RQ3:** What is the influence of CI process characteristics?
  - **RQ3.1**: What is the influence of the inter-commit time?
  - **RQ3.2**: What is the influence of the cycle size?

4.4 Subjects

We ran experiments on six Java subjects from the open-source Apache Commons project, which use Maven as build system.[8] They have been selected based on their size (>10*KLoC*), number of latest "working" commits[9] (>100), and mean number of test targets per commit (>5). All the selected projects come equipped with the developer's test suite. Table 3 lists the subjects and their characteristics.

Table 3: Subjects

Subject	*SHA*	*#Commits*	*KLoC*	*#Targets*	*#Tests*
Codec	5a9c79f	614	14.8	39	403
Compress	66338dd	627	34.5	94	475
Imaging	9c5dc5b	376	40.3	79	90
Io	f9e08f8	387	28.6	95	1014
Lang	ce0c082	521	77.8	163	3899
Math	71fd124	111	186.7	497	4864

N.B.: SHA, KLoC, #Targets, and #Tests refer to the first commit.

4.5 Evaluation metrics

To evaluate the ranking in an algorithm-independent way (e.g., RL-based and LTR; classification- and regression-based), we leverage the Fault Percentile Average (FPA) used in the LTR task for software defect prediction [35]. We define the *Rank Percentile Average* (*RPA*) to adapt the FPA to the prioritization problem: it is used to compute how much a *predicted* ranking is close to the *actual* ranking. The metric can evaluate a ranking independently of the specific testing criteria (e.g., fault detection). Let us assume that priority scores are increasing integers, from 1 to k, with k being the number of test targets to prioritize – a higher score means higher priority. Let us denote with $r_i = i$ the actual (true) ranking score of test target i (e.g., $R = \{5, 4, 3, 2, 1\}$, for $k = 5$ targets), and their sum with $r = r_1 + r_2 + \cdots + r_k = k(k+1)/2$ ($r = 15$ in the example). The prediction task produces a permutation of R (e.g., $R' = \{5, 2, 3, 4, 1\}$, with the second and fourth elements swapped). The following ratio represents the "proportion" of the *actual* ranking scores "contained" in the top m *predicted* test targets with respect to r:

$$\frac{1}{r}\sum_{i=k-m+1}^{k} r_i = \frac{1}{k(k+1)/2}\sum_{i=k-m+1}^{k} r_i. \quad (2)$$

For instance, for the top 2 elements of the predicted ranking R', the proportion is 7 over 15. The actual ranking R scores 9 over 15. *RPA* is defined as the average of this proportion over the k elements:

$$RPA = \frac{1}{k}\sum_{m=1}^{k}\frac{1}{r}\sum_{i=k-m+1}^{k} r_i = \frac{\sum_{m=1}^{k}\sum_{i=k-m+1}^{k} r_i}{k^2(k+1)/2}. \quad (3)$$

The higher the *RPA*, the better. The maximum value, RPA_M, is reached when the predicted ranking is equal to the actual one (r_i=i in the summation). With few manipulations, it can be shown that:

$$RPA_M = 1 - \frac{\sum_{i=1}^{k-1}(k-i)(k-i+1)}{k^2(k+1)} \quad (4)$$

which is less than 1. We normalize *RPA*, so that $NRPA \in [0, 1]$, computing the *Normalized-Rank-Percentile-Average*:

$$NRPA = \frac{RPA}{RPA_M}. \quad (5)$$

For our example, the predicted ranking R' has $RPA = 51/75 = 0.68$, RPA_M=55/75=0.73, and $NRPA$=0.68/0.73= 0.93. The $(N)RPA$ metric has an intuitive interpretation, representing the average of "how much" of the optimal-ranking scores is "contained" in the top-m predicted-ranking's objects. Moreover, the metric is more accurate than the Spearman's correlation coefficient on partial lists. For instance: consider an optimal ranking of 10 tests ($r_i = 10 \ldots 1$), and a partial ranking evaluation of the first three tests; assume we have: *predicted ranking 1* = 10, 2, 9 and *predicted ranking 2* = 10, 8, 9 (*optimal ranking*=10, 9, 8): the Spearman correlation coefficient is 0.5 in both cases, as it considers only the rank (3, 1, 2 in both cases), while RPA considers also the magnitude (selecting number 2 rather than 8 as second test) and gives, respectively, 0.07 and 0.1.

To investigate differences in the performance of algorithms in time-constrained scenarios, we consider the cases in which not all the selected test targets can be run at each cycle, e.g., because the inter-commit time is short compared to the tests execution time. We adopted the same constraints used in previous work [3] investigating the effect of time-constraints on regression testing, i.e., 25%, 50% and 75% of the number of selected test targets. In such cases, the *RPA* metric makes no sense: sublists of the optimal and the predicted rankings generally do not contain the same tests, and cannot be compared.

Considering the two prioritization criteria (fault detection and test execution time), at each commit we compute: the difference between the *predicted total tests execution time* (for the predicted ranking) and the *optimal total tests execution time* (for the optimal ranking); the difference between the *predicted number of failing tests* (for the predicted ranking) and the *optimal number of failing tests* (for the optimal ranking), with reference to the 25%, 50% and 75% lists. The larger these differences, the worse the predicted ranking. It should be noted that, as fault detection metric, we discard the well-known *APFD*, as it considers cumulative fault detection over time and is not appropriate in CI where the focus is on obtaining

[8] https://commons.apache.org.

[9] "Working" means not requiring deprecated libraries or old software versions (e.g., old Java, Maven) to build successfully.

feedback on individual test cases rather than on a whole test suite [5]. Additionally, in CI, changes in each release are very limited with respect to the codebase, which makes the number of faults per release very small (as in our dataset), compared to traditional processes [9]. This would lead to undetermined *APFD* values.

To quantify the cost, we compute in each cycle: the time for test selection, the time for prioritization (made up of *training time* and *ranking time*) and the time for the execution of the selected tests. Their sum is referred to as *end-to-end time*. The above metrics are computed at every commit, when tests are prioritized. Algorithms are run 30 times on each subject.

5 RESULTS

Table 4 shows the results of *test selection*. The first two rows report the *total number of selected targets* (test classes) and total number of *tests* of the selected targets across all the commits, and their *average per commit*. Rows 3 and 4 list the total number and percentage of failing targets and tests over selected ones.[10] The last two rows report the sum of the times for test selection (including time to check for changes upon a commit, to update the dependency graph and to extract the test classes) and for execution of all selected tests. The list of selected test targets at each commit, along with their execution time, the number and percentage of failed tests per target, and the metrics per target (cf. with Table 1) correspond to an instance for the ML-based prioritization algorithm, completed by the actual (during training) or predicted (in prioritization) ranking.

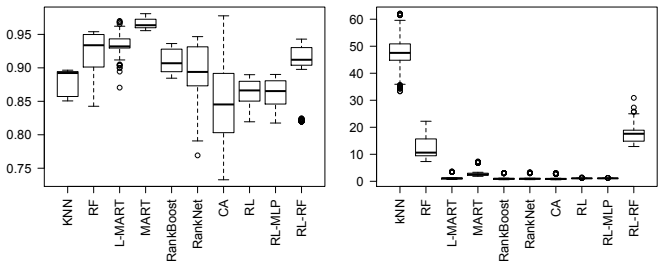
Table 4: Results of test selection

Subjects:	Codec	Compress	Imaging	Io	Lang	Math
# Selected targets	3,167 Av.: 5.16	10,724 Av.: 17.10	4,883 Av.: 12.98	5,299 Av.: 13.69	11,228 Av.: 21.55	3,926 Av.: 35.37
# Selected tests	46,864 Av.: 14.79	90,493 Av.: 144.32	20,086 Av.: 53.42	96,619 Av.: 249.66	389,404 Av.: 747.41	49,533 Av.: 446.24
# Failing targets	3 0.0947%	21 0.1958%	2 0.0409%	13 0.2453%	6 0.0534%	8 0.2037%
# Failing tests	3 0.0064%	32 0.0353%	2 0.0099%	16 0.0165%	16 0.0041%	9 0.0181%
Total sel. time (s)	1,096 Av.: 1.78	2,284 Av.: 3.64	2,106 Av.: 5.60	1,117 Av.: 2.88	2,910 Av.: 5.58	1,050 Av.: 9.46
Total exec. time (s)	941 Av.: 1.53	1,810 Av.: 2.89	2,739 Av.: 7.28	8,506 Av.: 21.98	1,367 Av.: 2.62	1,116 Av.: 10.06

5.1 RQ1: Prioritization Effectiveness and Cost

5.1.1 RQ1.1: Algorithms comparison. The *RPA* values and *ranking times* per algorithm are computed after prioritization at each commit. We consider the averages over commits of these values, obtaining 180 observations (6 subjects x 30 repetitions): these are shown in the boxplots in Fig. 2. The boxplots per subject are available in the supplemental material.[7]

Fig. 3 reports the total training times per algorithm, in logarithmic scale, averaged over subjects and repetitions. RTL algorithms take much longer, since training is repeated at each commit. However, RTL can rank test cases since the beginning: the *time to first prioritization* - *TTFP* - is null, whereas LTR must wait for training to finish (*TTFP* equals the training time). The test selection and



(a) *RPA* per commit (b) Ranking time (ms) per commit

Figure 2: Average RPA and ranking time per algorithm

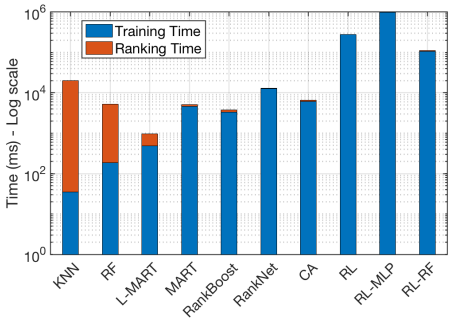


Figure 3: Average training and ranking times per algorithm

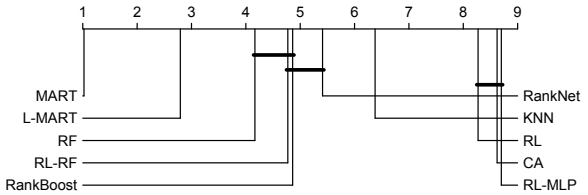
execution times (Table 4) plus the algorithm-dependent training and ranking times for prioritization make the *end-to-end time*.

We run one-way analysis of variance (ANOVA) considering the *algorithm* as factor and *RPA* and ranking time as response variables. The levels are grouped in order to show the differences as formulated in RQ1.1 to RQ1.3. To choose the statistical test for ANOVA, we checked the data for normality and for homoscedasticity by means of, respectively, the Shapiro-Wilk test and the Levene's test: the two null hypotheses of data coming from a normal distribution and of variances being homogeneous are both rejected at *p-value* <2.2E-16, which is the minimum value of the R statistical tool that we used. Therefore, we conducted a non-parametric ANOVA by means of the Friedman test, which is robust to non-normality and heteroscedasticity, with the Iman and Davemport extension [11].[11] The test detects if at least one factor's level significantly differs from another. We then run a *post hoc* test to detect what levels are different, by using the Shaffer's static method, a powerful method for *all pairwise* comparison exceeding nine algorithms [2].

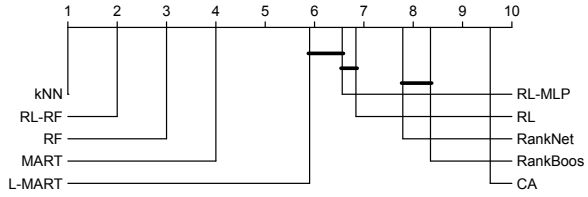
The test confirms, for the *RPA* and the ranking time (*p-value* <2.2E-16 in both cases), that there is at least one significant difference among the algorithms. Fig. 4a and Fig. 4b report the pairwise comparison results by the ranking plot, an adaptation of the Nemenyi's test critical difference plot working with other tests [2]: algorithms with no significant difference are grouped together using a bold horizontal line – the more distant two algorithms are (the distance being the average ranking), the smaller the *p-value* for the null hypothesis of equal performance.

[10] Note the small number of failing tests for all subjects, causing highly unbalanced datasets. Sensitivity of algorithms to this aspect is assessed in Section 5.2.2.

[11] The Iman and Davemport extension is a popular choice to improve the too conservative Friedman's statistic, suited to compare more than five algorithms [6].

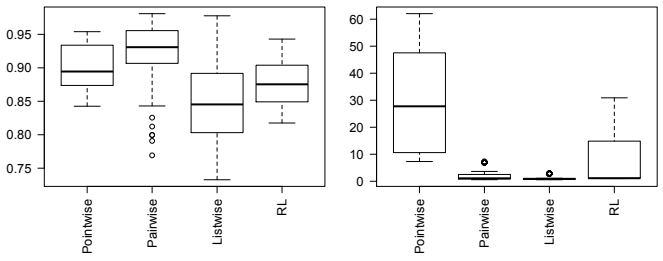


(a) *RPA*: the leftmost one has biggest *RPA*



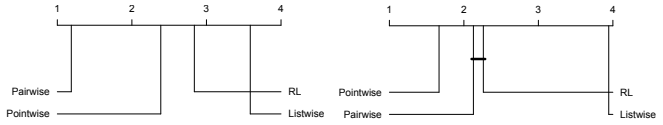
(b) *Ranking Time*: the leftmost has longest time

Figure 4: Pairwise comparison of algorithms



(a) RPA per commit **(b) Ranking time per commit (ms)**

Figure 5: Average performance of the four strategies



(a) RPA **(b) Ranking time**

Figure 6: Pairwise comparison of the four strategies

The algorithms performing better in terms of *RPA* are MART and its extension LambdaMART, followed by RF, RL-RF and RankBoost (with no significant differences). Their commonality is to be based on (decision or regression) trees, while most of others are based on neural networks (RankNet, RL-MLP, RL). The worst group includes RL-MLP, CA and RL, although their average *RPA* values are above 0.8. As for *ranking time*, KNN requires much longer, followed by RL-RF and RL. The best one is by far CA.

Summarizing: MART and L-MART have the best *RPA* and medium-level *ranking* and *training times*; CA performs poorer but is better for *ranking time* with a medium-level *training time*. KNN requires less *training time* than others LTR, but it takes long for ranking, not justifiably paid off by the *RPA*. RL algorithms require high training times, being online learning schemes; RL-RF has high ranking times, but good *RPA*. RL and RL-MLP have better times but poorer *RPA*.

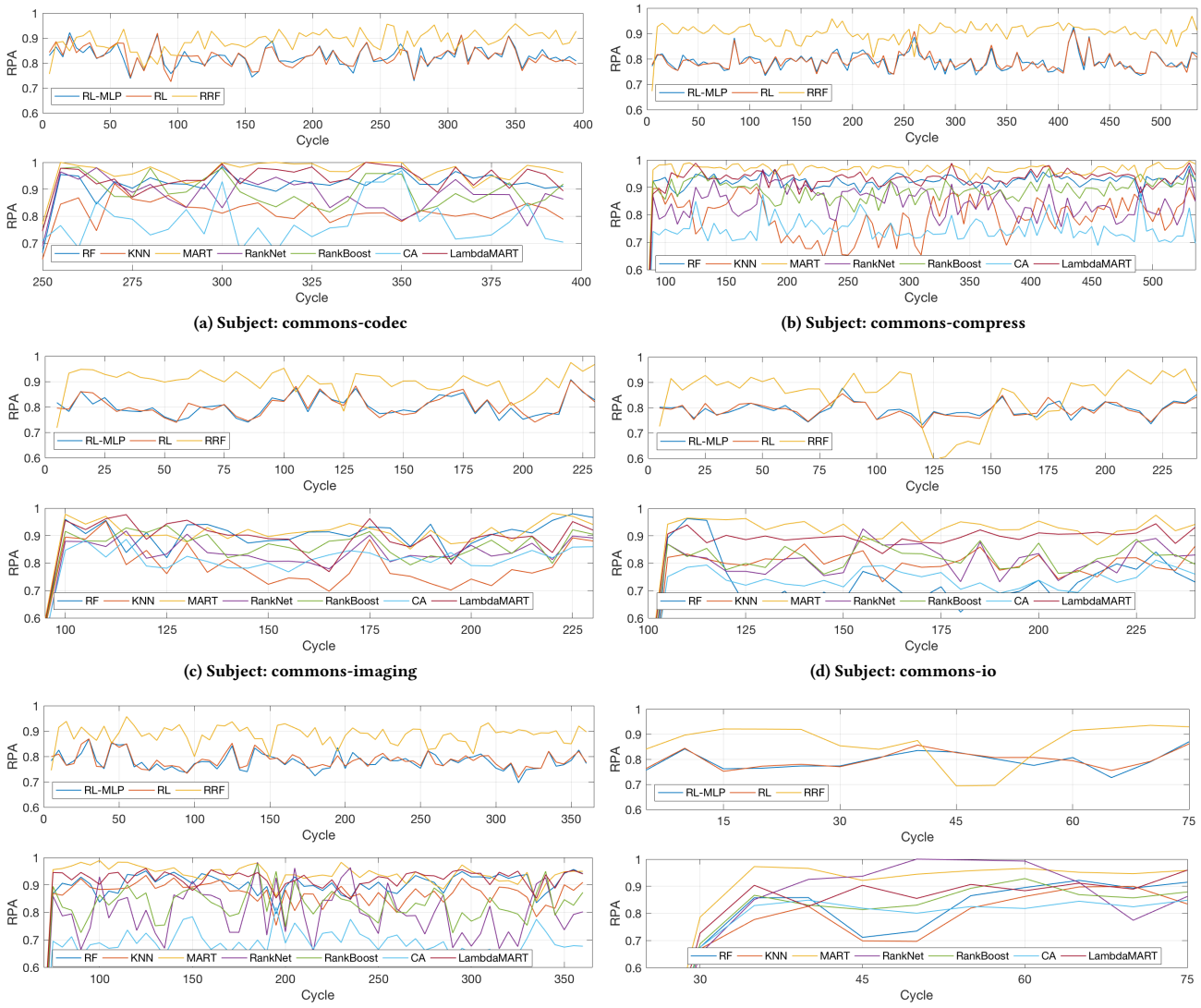
5.1.2 RQ1.2: Strategies comparison. We compare the four strategies (*RTL, LTR pointwise, LTR pairwise, LTR listwise)*, grouping the results of the respective algorithms. As the groups have an unequal number of algorithms, we run the Skillings-Mack test, an adaptation of the Friedman's statistic able of dealing with unbalanced designs [27]. The *p-value* is <2.2E-16 for both *RPA* and time. Fig. 5 and Fig. 6 show the boxplots and the ranking plots, respectively. In terms of *RPA*, the strategies differ significantly from each other, with pairwise algorithms being the best ones. In terms of ranking time, pointwise are the worst (because of KNN) and the listwise method (CA) is the best; RL and pairwise are comparable. Pointwise and pairwise LTR exhibit both higher *RPA* than RL methods: hence, online learning (by RL) does not necessarily ensure better performance than static methods, as seen in the analysis of single algorithms. This may depend on other features related to the code variability, as investigated in the next RQs.

5.1.3 RQ1.3: Categories comparison. We compare *non-ensemble vs ensemble* algorithms. The Mann-Whitney-Wilcoxon test is suited in this two-levels case. Again, the statistical test reports a significant difference, both in terms of *RPA* and prioritization time (*p-value* <2.2E-16 for both) – the averages *RPA* are 0.927 and 0.869 for *ensemble* and *non-ensemble* learning, respectively, and the average times are 7.206 ms and 10.376 ms. Thus, *ensemble* methods have both higher *RPA* and shorter ranking times. Looking at Fig. 3, they also have shorter training times.

5.2 RQ2: Influence of code under test

5.2.1 RQ2.1: CUT variability. Fig. 7 reports the *RPA* across the cycles (each *RPA* value is the average over 30 repetitions) for each subject/algorithm. For readability, the commits with one test class (that always gives *RPA*=1) are removed from the plots (hence less commits are shown than the actual ones) and the average *RPA* at each 5 commits is reported. The top subplot for each subject shows the *RTL* algorithms, the bottom one the *LTR* ones: in fact, while the latter ones require a training phase during which no prediction is carried out, the former ones do not require training and give predictions sooner, being based on reinforcement learning. The performance of learning algorithms depend on the representativeness of the learnt model: in highly variable CI scenarios (when the code changes a lot, and failure proneness and/or test execution time change too), ML predictive performance may be poor. Looking at the plots in Figure 7, the performance are quite stable, but some algorithms, like RL-RF and KNN, exhibit trends in some scenarios. It is interesting to investigate if and to what extent the trends are related to code variability. We proceed as follows.

We first run, for each subject's dataset, the *Principal Component Analysis* (PCA) over the features, so as to remove their first-order inter-correlation, and select a number of PCs necessary to keep the 95% of the original variance (for the 6 subjects, the number of PCs was: 18, 16, 18, 17, 17, 15). Then, we combine the PCs in one time series to have a synthetic indicator of the trend, capturing the variability of the code metrics: to this aim, we exploit the Hotelling's multivariate control charts [10]. Multivariate control charts are used to monitor two or more interrelated process variables in order to detect shifts in the mean or the relationship between several interrelated variables [24]. A Hotelling chart computes a statistic,



(a) Subject: commons-codec

(b) Subject: commons-compress

(c) Subject: commons-imaging

(d) Subject: commons-io

(e) Subject: commons-lang

(f) Subject: commons-math

Figure 7: Average RPA plots across cycles

named T^2, that combines the information from the dispersion and the mean of several related variables. It can be used in a one-phase setting (i.e., by using the same data that is being plotted on the control chart to characterize the normal behavior) or two-phase setting (i.e., using part of the data to characterize the normal behavior and the other part to be checked for deviations); our aim falls in the one-phase setting. Given the selected k PCs, the statistic for observation i is:

$$T_i^2 = (Y_i - \bar{Y})' S^{-1} (Y_i - \bar{Y}) \quad (6)$$

where Y_i is the vector of k measurements for observation i, $\bar{Y}$ is the vector of sample means of the k variables, S^{-1} is the inverse of the sample covariance matrix, which provides information regarding the relationship between different variables. Plots of T^2 are omitted for lack of space, but they are made available separately.[7]

Using the T^2 time series as metrics variability indicator and the *RPA* time series as performance indicator, we finally compute the *transfer entropy* (TE) between T^2 and *RPA*, namely the amount of directed transfer of information between the two time series. Differently from correlation, TE informs us about causality – it actually generalizes the Granger causality test – computing how much information about the transition between two consecutive steps of the *RPA* time series can be found in the past state of T^2. How "far" the past is depends on the user-defined *delay* between the two time series d. We compute the TE under 10 different delays, $d = 1$ to $d = 10$ and using the Kraskov-Stögbauer-Grassberge (KSG) TE estimator [14]. To capture any possible causality relations, we consider, beside the average, the maximum of the TE values over the d values, so as to identify the condition under which the T^2 affects more *RPA*. Table 5 reports the maximum TE per algorithm and subject over the 10 delays. The last column summarizes which algorithm has been

Table 5: Transfer entropy

Subjects:	Codec	Compress	Imaging	Io	Lang	Math	All
	$\text{Max}_d\,TE$	$\text{Max}_d\,TE$	$\text{Max}_d\,TE$	$\text{Max}_d\,TE$	$\text{Max}_d\,TE$	$\text{Max}_d\,TE$	$\text{Avg}(\text{Max}_d)$
KNN	0.0267	0.0302	0.0397	0.0297	0.0379	0.1018	0.0443
RF	0.0360	0.0319	0.0419	0.0230	0.0129	0.1256	0.0452
L-MART	0.0256	0.0308	0.0302	0.0430	0.0382	0.0804	0.0414
MART	0.0341	0.0229	0.0157	0.0325	0.0158	0.0854	0.0344
RankNet	0.0274	0.0304	0.0499	0.0486	0.0216	0.0766	0.0424
RankBoost	0.0533	0.0151	0.0298	0.0355	0.0350	0.0682	0.0395
CA	0.0540	0.0267	0.0577	0.0590	0.0380	0.0783	0.0523
RL	0.0336	0.0179	0.0428	0.0037	0.0177	0.0615	0.0295
RL-MLP	0.0373	0.0239	0.0313	0.0091	0.0317	0.0510	0.0307
RL-RF	0.0191	0.0119	0.0182	0.0256	0.0259	0.0891	0.0316

more sensitive to the variation of the T^2 time series; the ranking is as follows: *CA, RF, KNN, RankNet, LambdaMART, RankBoost, MART, RL-RF, RL-MLP, RL*: all *LTR* approaches are more affected by the T^2 code variability indicator, while all *RL-based* algorithms are more robust to variations, likely because of their online-learning nature. This suggests that in highly variable contexts, with intense metrics' changes, RL-based approaches can be preferred, to avoid having to re-train a static LTR algorithm too often. In contrast, static contexts can stress the good performances of LTRs.

5.2.2 RQ2.2: Code failure proneness. We created an artificial dataset from our original one, by injection of failing test case outcomes. In each commit, a changed class is selected randomly, along with its dependent classes. We considered the corresponding test classes, and altered the outcome of their tests form pass to fail with a given probability p (we used $p = 0.15$). The so-produced dataset is more balanced, with percentages of 6-7% of failing test targets (Table 6). Results are plotted in Fig. 8.

Table 6: Scenario with error injection

Subjects:	Codec	Compress	Imaging	Io	Lang	Math
Failing targets	388 [6.80%]	652 [6.08%]	165 [3.38%]	363 [6.85%]	785 [6.99%]	283 [7.20%]
Failing tests	3,186 [12.25%]	3,244 [3.58%]	846 [4.21%]	4,404 [4.55%]	14,396 [3.69%]	1,740 [3.51%]

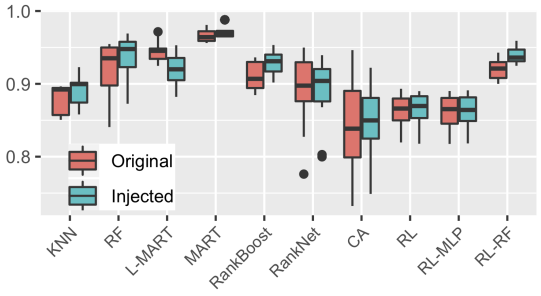


Figure 8: Comparison on failure proneness: RPA

The Wilcoxon signed ranked test is run on each pair (Table 7). Only 2 out of 10 algorithms turn out to be not affected by a greater number of failing tests. Those affected are, in decreasing order of confidence: LambdaMART, RF, RL-RF, RankBoost, KNN, MART, RL, RL-MLP. These are the three RTL algorithms and the LTR ones using *ensemble* learning, except KNN. As for RQ1.3, they perform generally better, but are more sensitive to the dataset balancing. *Non-ensemble* LTR algorithms (RankNet and CA) are less sensitive.

Table 7: Statistical analysis: scenario with error injection. Difference between averages (Inj: Injection scenario; Orig: original scenario, with no injection) and *p-value* for the null hypothesis H_0; the *RPA* of the two scenarios is the same

Algorithm	RL-MLP	RL	RL-RF	RF	kNN
Avg(Inj)-Avg(Orig)	1.25e-3	3.39e-3	1.86e-2	1.73e-2	1.21e-2
p-value	**9.46e-3**	**2.14e-7**	**1.67e-11**	**1.67e-11**	**1.42e-11**
Algorithm	MART	RankNet	RankBoost	CA	L-MART
Avg(Inj)-Avg(Orig)	5.30e-3	-1.87e-3	1.92-2	4.69e-3	-2-52e-2
p-value	**2.02e-8**	5.21e-1	**1.42e-11**	2.17e-1	**1.73e-11**

5.2.3 RQ2.3: Feature selection. When building a model, redundant or irrelevant features increase computational costs and can result in poor predictive performance. To identify the most important features in a CI context we used an unsupervised feature selection approach called Principal Feature Analysis (PFA) [20]. Basically, PFA exploits the structure of the principal components of the original feature set to select a subset that keeps most of the essential information. One advantage of PFA over other feature selection techniques is that it operates independently of any learning algorithm as it depends only on the original feature set.

Our implementation of PFA decides how many features should be selected by traditional PCA [31] to identify the principal components required to keep at least 90% of the cumulative explained variance. For all our subjects, 10 features were selected out of the original 50; hence, 20% of the features suffice to keep 90% of the original data variance. One feature among those listed in Table 1 was selected for all the subjects, which is *AvgEssential* –a cyclomatic complexity metric obtained after iteratively replacing all well structured control structures with single statements to account for any branches into or out of a loop or decision; other six size-related features were selected for at least 4 of our 6 subjects: *AvgCyclomatic*, *CountDeclFunction*, *CountDeclMethodDefault*, *CountLine*, *CountLineComment*, and *CountDeclMethod*. Overall, the selection was balanced across the different attributes types and features from the *program size*, *cyclomatic complexity*, and *object-oriented* groups were selected 38%, 33%, and 28% of the time, respectively.

5.3 RQ3: CI process characteristics

5.3.1 RQ3.1: Inter-commit time. RQ1 investigated algorithms' performance regardless of the time available to execute test cases. If time limits do not allow to run all selected tests, the tester might be interested in analyzing the algorithms' performance under various time constraints. Based on a previous work studying regression testing under different time constraints [3], we consider scenarios in which 25%, 50% and 75% of the number of selected tests can be run at each cycle, and assess the impact on ranking effectiveness. In large-scale systems, practitioners could look for more aggressive reductions; deeper exploration of this aspect is left to future work.

Table 8: Differences between predicted and optimal rankings in tests execution times (s) and number of failing tests (best values per columns are in bold, worst values in italics)

Algorithms	Time-constrained scenarios					
	25%		*50%*		*75%*	
	Time	Failures	Time	Failures	Time	Failures
KNN	0.5527	-2.0000	1.6622	-2.0000	4.8707	-1.3333
RF	0.9352	-2.6333	2.5995	-1.9333	4.4247	**-1.0500**
L-MART	0.2452	-3.1333	0.9877	-2.8000	2.4506	-1.7833
MART	**0.0164**	-2.4000	**0.0386**	-1.6667	**0.0193**	-1.6667
RankBoost	1.7471	**-1.3333**	2.5321	**-1.1667**	3.2461	-1.1667
RankNet	1.0548	-1.7333	2.4865	-1.3333	2.9609	-1.1167
CA	*2.5024*	-2.9600	*3.5722*	-2.0500	4.1488	-1.7500
RL	1.8893	-5.2167	3.4928	-3.6000	4.8379	-2.2167
RL-MLP	1.7781	-5.4500	3.4982	-3.9833	4.8348	-1.9667
RL-RF	0.5796	*-7.2000*	2.2932	*-4.5333*	*5.1311*	*-2.3500*

As discussed in Section 4.5, the *RPA* metric makes no sense in this case: Table 8 reports the differences between the predicted and optimal rankings, in terms of *tests execution time* (average over all subjects and commits) and *total number of failing tests* (average over all subjects). The former is always ≥ 0, the latter always ≤ 0: they are zero when the predicted ranking is equal to the optimal ranking in the top 25/50/75% list of tests. The bigger their absolute values, the worse the performance of the algorithm. Data per subject are made available separately.[7]

Whenever results are in line with the *RPA* (RQ1), e.g., L-MART and MART on test execution times or RankBoost on number of failures that are close to the optimal one, it means that mis-ranked tests had low impact (e.g., because their execution times or number of failures is not much different than those of the optimal ranking's tests). When results are not in line with *RPA*, e.g., RL-RF and RankNet on failures, then either the algorithm performs well for one prioritization criterion and bad for the other and/or the tests that were mis-ranked, even by a small amount, had a high impact.

Clearly, short inter-commit time may influence the choice of the algorithm to adopt more than the RPA metric.

5.3.2 RQ3.2: History length. We investigate whether and to what extent the amount of history used for learning impacts performance of the algorithms. For LTR, this history is used just once at the beginning for training; for RTL, the history is in a sliding window (the memory) used to update the learning process online. Fig. 9 shows the average *RPA* and *training time* over all the commits and projects *vs* four values of training sample size.

The *RPA* is quite insensitive with respect to a training sample size bigger than 500 observations, for all the algorithms. The average good performance of LTR even with smaller training sets mitigates the drawback of not having predictions during the training process. Contrarily, the training time expectedly increases remarkably with training sample size (the graph is in logarithmic scale): the RTL algorithms are severely affected, followed by pairwise and listwise, and finally, the least affected ones, the pointwise algorithms.

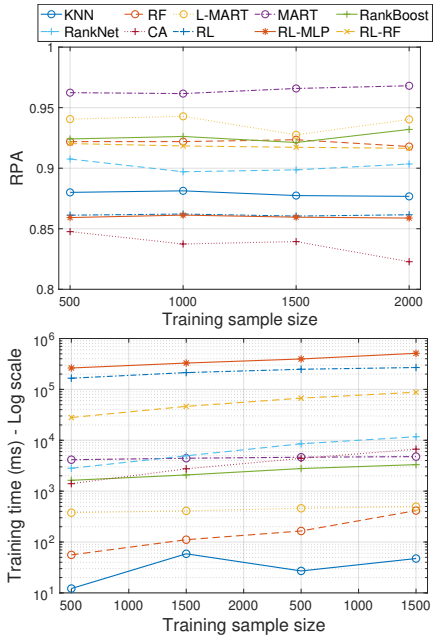


Figure 9: Sensitivity of RPA and training time to training sample size

6 DISCUSSION

6.1 Threats to validity

Threats to construct validity may descend from the adopted evaluation metrics (Section 4.5). We used a new metric, *RPA*, to assess the algorithms' performance: with a different metric, RQ1 might have received different answers. The same threat may affect the cost-related analyses. Indeed, there do not exist other studies comparing (heterogeneous) ML techniques for test prioritization, so we could not rely on established metrics for a fair comparison. Another threat might be due to our identification of characteristics of CUT (RQ2) or CI process (RQ3): if our experimental design does not properly capture the relevant features or their influence on regression testing performance, our answers to RQ2 and RQ3 might not be valid. To mitigate these risks, other studies should be conducted.

As for *threats to internal validity*, one risk may derive from inappropriate settings of the tools we used for the ML algorithms (Section 4.1). Different settings or also different choices for the values of parameters of the experiment might have produced different results. To control this threat, we performed sensitivity analyses to our feature selection choices; however, due to the complexity of our assessments, we cannot exclude that some decisions in the experiment design might have biased our comparisons, by impacting differently the algorithms. Another threat concerns our study about failure proneness, in which we artificially injected failing test outcomes. Real faults in production might produce different effects: however this is a procedure commonly used for reliability studies and we opted for this as we did not have real faults.

As for *threats to external validity*, our experiments were run on only six Java subjects from the Apache library. Although these six projects are very active, have a large contributors' base and a long history, they might not be representative of industrial practice. They

Table 9: ML-based prioritization requirements *vs* decision variables. The ranks are from 1st (best) to 10th (worst)

	Prioritization effectiveness (*RPA*)	Ranking time	Training time	Online	Robustness to code variability	Robustness to #failing tests	Robustness to training size (*RPA*)	Robustness to training size (time)	Time-constrained effectiveness (50%) (time, failures)
Pointwise	2nd	4th	1st	No	8th, 9th	1st, 6th	2nd	1st	2nd, 2nd
Pairwise	1st	3th	2nd	No	4th, 5th, 6th, 7th	5th, 7th, 9th, 10th	3rd	2nd	1st, 1st
Listwise	4th	1th	3rd	No	10th	2nd	4th	3rd	4th, 3rd
RTL	3rd	2nd	4th	Yes	1th, 2nd, 3rd	3rd, 4th, 8th	1th	4th	3rd, 4th
Non-ensemble	2nd	2nd	2nd	Can be	1th, 3rd, 4th, 9th, 10th	1th, 2nd, 3rd, 4th, 6th	2nd	1th	2nd, 2nd
Ensemble	1th	1th	1th	Can be	2nd, 5th, 6th, 8th	5th, 7th, 8th, 9th ,10th	1th	2nd	1th, 1th
KNN	7th	10th	1th	No	8th	6th	4th	1th	3rd, 5th
RF	3rd	8th	2nd	No	9th	9th	5th	3rd	7th, 4th
L-MART	2nd	6th	3rd	No	6th	10th	10th	2nd	2nd, 7th
MART	1th	7th	5th	No	4th	5th	6th	4th	1th, 3rd
RankBoost	5th	2nd	4th	No	5th	7th	8th	5th	6th, 1th
RankNet	6th	3rd	7th	No	7th	1th	7th	7th	5th, 2nd
CA	9th	1th	6th	No	10th	2nd	9th	6th	10th, 6th
RL	8th	4th	9th	Yes	3rd	4th	1th	9th	8th, 8th
RL-MLP	10th	5th	10th	Yes	2nd	3rd	3rd	10th	9th, 9th
RL-RF	4th	9th	8th	Yes	1th	8th	2nd	8th	4th,10th

are also not enough representative for generalizing our conclusions to programs developed with other languages or processes (not open source). To control this threat, more studies should be conducted. Hopefully, the fact that we explain in detail our experimental settings and make available code and data for replication, will support other researchers in performing replication studies over different subjects and contexts, even using different ML algorithms.

6.2 Guidelines

Results led us to draw the following suggestions for applying Machine Learning algorithms to regression testing in CI:

- A valuable approach under short inter-commit times is to run a ML prioritization algorithm after selection, so that it acts on a small problem size and on test cases relevant for that commit. An incremental static selection approach like the one we experimented appears suited for the time requirements of CI.
- The testing criteria should be identified first, along with their relative weight within the ranking function: they determine what the *optimal ranking* is. If a coarse-grain ranking is enough, then classification algorithms may work well; otherwise, regression approaches better manage fine-grain ranking problems. Most of the experimented algorithms can work with both formulations.
- The ML algorithm's input consists of the *features* more likely related to the identified testing criteria. We targeted feature selection in an algorithm-independent way (unsupervised), first defining relevant features and then choosing the algorithm based on other requirements – this simplifies the problem. An alternative is to run feature selection for each potential algorithm being considered, so as to infer the best features for each of them: this, although more precise, can be much more time-consuming.
- In the plethora of ML algorithms to choose from, a suggestion is to first decide the learning *strategy* and, possibly, the *category* (ensemble or non-esemble), and then opt for the *specific algorithm*. The choice depends on the *requirements* for prioritization, in terms of desired ranking effectiveness and efficiency (RQ1), tolerable sensitivity to code features (RQ2), and on the CI process features (RQ3). They are strongly dependent on the CI context.

Table 9 reports the non-exhaustive list of requirements we have investigated in this article along with the "values" of the decision variables (algorithm, strategy, category) ranked from best to worst according to the reported results. The lowest part of Table 9 suffices to drive the choice among the algorithms we investigated; for other algorithms, the first and second parts may be used. The choice of the strategy/algorithm depends on the relative importance given to the requirements in the specific CI context.

A final suggestion concerns the tuning of algorithm parameters. In this study, we adopted default parameter values for fair comparison, but (as hinted already in validity threats) performance can vary significantly depending on tuning. This is especially true for the individual algorithms, while the choice of the strategy or category is less affected. A tuning step with any existing methods (e.g., grid or randomized search) may be in order.

7 CONCLUSIONS

Continuous Integration practices in large industrial settings pose specific requirements on test selection and prioritization for regression testing, due to the frequent commits, with short inter-commit times and few changes with respect to the codebase size.

This study has presented a comprehensive evaluation of ten machine learning algorithms for test prioritization after selection in CI. Based on the results of controlled experiments with open source subjects – including an analysis of features of the code under test and of the test process influencing algorithms' performance - guidelines have been devised for testers to select and tune the ML algorithms best fitting their needs.

ACKNOWLEDGMENTS

This work has been partially supported by the PRIN 2015 project "GAUSS" funded by MIUR, and partially supported by INES 2.0, FACEPE grants PRONEX APQ 0388-1.03/14 and APQ-0399-1.03/17, CAPES grant 88887.136410/2017-00, and CNPq grant 465614/2014-0.

REFERENCES

[1] Benjamin Busjaeger and Tao Xie. 2016. Learning for Test Prioritization: An Industrial Case Study. In *24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*. ACM, New York, NY, 975–980. https://doi.org/10.1145/2950290.2983954

[2] Borja Calvo and Guzman Santafe. 2015. scmamp: Statistical Comparison of Multiple Algorithms in Multiple Problems. *The R Journal* 8, 1 (2015), 248–256.

[3] Hyunsook Do, Siavash Mirarab, Ladan Tahvildari, and Gregg Rothermel. 2008. An empirical study of the effect of time constraints on the cost-benefits of regression testing. In *16th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*. ACM, 71–82. https://doi.org/10.1145/1453101.1453113

[4] Vinicius H. S. Durelli, Rafael S. Durelli, Simone S. Borges, Andre T. Endo, Marcelo M. Eler, Diego R. C. Dias, and Marcelo P. Guimarães. 2019. Machine Learning Applied to Software Testing: A Systematic Mapping Study. *IEEE Transactions on Reliability* 68, 3 (2019), 1189–1212. https://doi.org/10.1109/TR.2019.2892517

[5] Sebastian Elbaum, Gregg Rothermel, and John Penix. 2014. Techniques for Improving Regression Testing in Continuous Integration Development Environments. In *22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*. ACM, New York, NY, 235–245. https://doi.org/10.1145/2635868.2635910

[6] Salvador García, Alberto Fernández, Julián Luengo, and Francisco Herrera. 2010. Advanced nonparametric tests for multiple comparisons in the design of experiments in computational intelligence and data mining: Experimental analysis of power. *Information Sciences* 180, 10 (2010), 2044 – 2064. https://doi.org/10.1016/j.ins.2009.12.010

[7] Milos Gligoric, Lamyaa Eloussi, and Darko Marinov. 2015. Practical Regression Test Selection with Dynamic File Dependencies. In *2015 International Symposium on Software Testing and Analysis (ISSTA)*. ACM, New York, NY, 211–222. https://doi.org/10.1145/2771783.2771784

[8] Alireza Haghighatkhah, Mika Mäntylä, Markku Oivo, and Pasi Kuvaja. 2018. Test prioritization in continuous integration environments. *Journal of Systems and Software* 146 (2018), 80 – 98. https://doi.org/10.1016/j.jss.2018.08.061

[9] H. Hemmati, Z. Fang, and M. V. Mantyla. 2015. Prioritizing Manual Test Cases in Traditional and Rapid Release Environments. In *IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 10. https://doi.org/10.1109/ICST.2015.7102602

[10] H. Hotelling. 1947. Multivariate quality control. In *Techniques of Statistical Analysis*, Wallis W.A. Eisenhart C., Hastay M. (Ed.). McGraw-Hill, New York (1947), 111–184.

[11] Ronald L. Iman and James M. Davenport. 1980. Approximations of the critical region of the fbietkan statistic. *Communications in Statistics - Theory and Methods* 9, 6 (1980), 571–595. https://doi.org/10.1080/03610928008827904

[12] Muhammad Khatibsyarbini, Mohd Adham Isa, Dayang N.A. Jawawi, and Rooster Tumeng. 2018. Test case prioritization approaches in regression testing: A systematic literature review. *Information and Software Technology* 93 (2018), 74–93. https://doi.org/10.1016/j.infsof.2017.08.014

[13] Eric Knauss, Miroslaw Staron, Wilhelm Meding, Ola Söder, Agneta Nilsson, and Magnus Castell. 2015. Supporting Continuous Integration by Code-churn Based Test Selection. In *IEEE/ACM 2nd International Workshop on Rapid Continuous Software Engineering (RCoSE)*. IEEE, 19–25. https://doi.org/10.1109/RCoSE.2015.11

[14] Alexander Kraskov, Harald Stögbauer, and Peter Grassberger. 2004. Estimating mutual information. *Phys. Rev. E* 69 (2004), 066138. Issue 6. https://doi.org/10.1103/PhysRevE.69.066138

[15] Remo Lachmann, Sandro Schulze, Manuel Nieke, Christoph Seidl, and Ina Schaefer. 2016. System-level test case prioritization using machine learning. In *15th IEEE International Conference on Machine Learning and Applications*. IEEE, 361–368.

[16] Owolabi Legunsen, Farah Hariri, August Shi, Yafeng Lu, Lingming Zhang, and Darko Marinov. 2016. An Extensive Study of Static Regression Test Selection in Modern Software Evolution. In *24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*. ACM, New York, NY, 583–594. https://doi.org/10.1145/2950290.2950361

[17] Alexandre R. Lenz, Aurora Pozo, and Silvia R. Vergilio. 2013. Linking software testing results with a machine learning approach. *Engineering Applications of Artificial Intelligence* 26, 5 (2013), 1631–1640. https://doi.org/10.1016/j.engappai.2013.01.008

[18] Hang Li. 2011. *Learning to Rank for Information Retrieval and Natural Language Processing*. Morgan & Claypool, San Rafael, CA.

[19] Long-Ji Lin. 1992. Self-improving reactive agents based on reinforcement learning, planning and teaching. *Machine Learning* 8, 3 (01 May 1992), 293–321. https://doi.org/10.1007/BF00992699

[20] Yijuan Lu, Ira Cohen, Xiang Sean Zhou, and Qi Tian. 2007. Feature selection using principal feature analysis. In *15th ACM International Conference on Multimedia (MM)*. ACM, New York, NY, 301–304. https://doi.org/10.1145/1291233.1291297

[21] Mike McGarr, Dianne Marsh, and the Developer Productivity team. 2017. Towards true continuous integration: distributed repositories and dependencies. https://medium.com/netflix-techblog/towards-true-continuous-integration-distributed-repositories-and-dependencies-2a2e3108c051

[22] Atif Memon, Zebao Gao, Bao Nguyen, Sanjeev Dhanda, Eric Nickell, Rob Siemborski, and John Micco. 2017. Taming Google-Scale Continuous Testing. In *IEEE/ACM 39th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*. IEEE, 233–242.

[23] Ade Miller. 2008. A Hundred Days of Continuous Integration. In *Agile 2008 Conference*. IEEE, 289–293. https://doi.org/10.1109/Agile.2008.8

[24] Douglas C. Montgomery. 1997. *Introduction to statistical quality control* (3 ed.). Wiley, New York, NY.

[25] Y. Pang, X. Xue, and A. S. Namin. 2013. Identifying Effective Test Cases through K-Means Clustering for Enhancing Regression Testing. In *12th International Conference on Machine Learning and Applications*. IEEE, 78–83. https://doi.org/10.1109/ICMLA.2013.109

[26] Naren Ramakrishnan. 2009. The Pervasiveness of Data Mining asnd Machine Learning. *Computer* 42, 8 (Aug 2009), 28–29. https://doi.org/10.1109/MC.2009.268

[27] John H. Skillings and Gregory A. Mack. 1981. On the Use of a Friedman-Type Statistic in Balanced and Unbalanced Block Designs. *Technometrics* 23, 2 (1981), 171–177. http://www.jstor.org/stable/1268034

[28] Helge Spieker, Arnaud Gotlieb, Dusica Marijan, and Morten Mossige. 2017. Reinforcement Learning for Automatic Test Case Prioritization and Selection in Continuous Integration. In *26th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. ACM, New York, NY, 12–22. https://doi.org/10.1145/3092703.3092709

[29] Daniel Ståhl and Jan Bosch. 2013. Experienced benefits of continuous integration in industry software product development: A case study. In *12th IASTED International Conference on Software Engineering*. ACTA Press, Calgary, 736–743.

[30] Stephen W. Thomas, Hadi Hemmati, Ahmed E. Hassan, and Dorothea Blostein. 2014. Static Test Case Prioritization Using Topic Models. *Empirical Software Engineering* 19, 1 (2014), 182–212. https://doi.org/10.1007/s10664-012-9219-7

[31] Michael E Tipping and Christopher M Bishop. 1999. Probabilistic principal component analysis. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)* 61, 3 (1999), 611–622.

[32] Paolo Tonella, Paolo Avesani, and Angelo Susi. 2006. Using the Case-Based Ranking Methodology for Test Case Prioritization. In *22nd IEEE International Conference on Software Maintenance (ICSM)*. IEEE, 10. https://doi.org/10.1109/ICSM.2006.74

[33] Hado Philip van Hasselt. 2012. Reinforcement Learning in Continuous State and Action Spaces. In *Reinforcement Learning. Adaptation, Learning, and Optimization*, M. Wiering and M. van Otterlo (Eds.), Vol. 12. Springer, Berlin, Heidelberg, 207–251. https://doi.org/10.1007/978-3-642-27645-3_7

[34] Ting Wang and Tingting Yu. 2018. A Study of Regression Test Selection in Continuous Integration Environments. In *29th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 135–143. https://doi.org/10.1109/ISSRE.2018.00024

[35] X. Yang, K. Tang, and X. Yao. 2015. A Learning-to-Rank Approach to Software Defect Prediction. *IEEE Transactions on Reliability* 64, 1 (2015), 234–246. https://doi.org/10.1109/TR.2014.2370891

[36] Shin Yoo and Mark Harman. 2012. Regression testing minimization, selection and prioritization: a survey. *Software Testing, Verification and Reliability* 22, 2 (2012), 67–120. https://doi.org/10.1002/stvr.430

[37] Du Zhang and Jeffrey J.P. Tsai. 2003. Machine Learning and Software Engineering. *Software Quality Journal* 11, 2 (2003), 87–119. https://doi.org/10.1023/A:1023760326768