Machine Learning Regression Techniques for Test Case Prioritization in Continuous Integration Environment

Enrique A. da Roza, Jackson A. Prado Lima, Rogério C. Silva, Silvia Regina Vergilio
Department of Computer Science, Federal University of Paraná, Curitiba, PR, Brazil
Emails: earoza@inf.ufpr.br, japlima@inf.ufpr.br, rcsilva@inf.ufpr.br, silvia@inf.ufpr.br

2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER) | 978-1-6654-3786-8/22/$31.00 ©2022 IEEE | DOI: 10.1109/SANER53432.2022.00034

***Abstract*—Test Case Prioritization (TCP) techniques are a key factor in reducing the regression testing costs even more when Continuous Integration (CI) practices are adopted. TCP approaches based on failure history have been adopted in this context because they are more suitable for CI environment constraints: test budget and test case volatility, that is, test cases may be added or removed over the CI cycles. Promising approaches are based on Reinforcement Learning (RL), which learns with past prioritization, guided by a reward function. In this work, we introduce a TCP approach for CI environments based on the sliding window method, which can be instantiated with different Machine Learning (ML) algorithms. Unlike other ML approaches, it does not require retraining the model to perform the prioritization and any code analysis. As an alternative for the RL approaches, we apply the Random Forest (RF) algorithm and a Long Short Term Memory (LSTM) deep learning network in our evaluation. We use three time budgets and eleven systems. The results show the applicability of the approach considering the prioritization time and the time between the CI cycles. Both algorithms take just a few seconds to execute. The RF algorithm obtained the best performance for more restrictive budgets compared to the RL approaches described in the literature. Considering all systems and budgets, RF reaches Normalized Average Percentage of Faults Detected (NAPFD) values that are the best or statistically equivalent to the best ones in around 72% of the cases, and the LSTM network in 55% of them. Moreover, we discuss some implications of our results for the usage of the algorithms evaluated.**

***Index Terms*—Recurrent Neural Networks, Machine Learning, Continuous Integration, Regression Testing**

I. Introduction

Continuous Integration (CI) is a common practice in the software industry [1]. To support this practice, the usage of CI environments is a key factor to allow the integration of changes quickly and in a cost-effective way. As the software evolves, new features are introduced, faults are removed, and many changes occur. To ensure that these new changes do not produce an expected behavior in the software, and that this software produces the same previous outputs, regression testing is performed.

However, the number of test cases tends to grow, making the execution of the existing test sets expensive. Furthermore, it should be considered that CI environments need to work with a limit amount of resources to perform such tests, called test budget. To minimize the resources used, a popular and widely adopted technique is the prioritization of test cases [2]. The *Test Case Prioritization (TCP)* problem aims at finding the best execution order for the test cases that satisfies certain specific criteria, such as early fault detection.

There are many TCP approaches in the literature [3]. The most popular in CI environments uses historical data [4]. This approach assumes that test cases that failed in previous builds tend to fail again in the future. Approaches that require exhaustive analysis are costly and inefficient [4]; the time available to run the prioritized test suite can be reduced if prioritization takes too long [5]. As a consequence, there is an effort to create lightweight approaches for Test Case Prioritization in Continuous Integration environment (TCPCI). In this sense, some pieces of work have explored *Machine Learning (ML)* techniques. *Reinforcement Learning (RL)* based approaches have presented promising results and present some advantages in the CI context [6]. These approaches observe previous cycles and learn with past prioritizations guided by a reward function (online learning). They better adapt to the dynamic nature of the CI environment and consider test case volatility, a characteristic associated with the fact that test cases may be added and/or removed over the cycles.

We can mention two approaches that can be considered the state-of-the-art in CI: i) `RETECS` [7] (*Reinforced Test Case Selection*): uses an agent, for instance, an *Artificial Neural Network (ANN)*, to interact with the CI environment. Based on the environment *state*, the agent defines an *action* (prioritization) to be applied in such an environment. The *state* is given by the information about a test case, such as the test case duration, historical failure data, and previous last execution. After, according to its previous action's performance, the agent receives a reward (feedback). Based on rewards provided by a reward function, the agent adapts its experience for future actions; and ii) `COLEMAN` [8] (***Combinatorial VOlatiLE Multi-Armed BANdit***): in the *Multi-Armed Bandit (MAB)* scenario, a player plays on a set of slot machines (or arms/actions) that even identical produce different gains. A test case is considered an arm. After a player pulls one of the arms in a turn, a reward is received drawn from some unknown distribution, thus aiming to maximize the sum of the rewards. Different strategies, called MAB policies, can be used to choose the next arm by observing previous rewards and decisions.

978-1-6654-3786-8/22/$31.00 ©2022 IEEE
DOI 10.1109/SANER53432.2022.00034

Both approaches work with the *Sliding Window* (*SW*) method [9]. RETECS uses a memory representation (*sliding window*) to delimit how much past information is used to learn. FRRMAB, a MAB policy adopted by COLEMAN, works with a sliding window as a smoother way to consider dynamic environments, allowing the observation of the changes in the quality of the arms (test cases) along the search process.

Considering that the use of a *SW* allows evaluating a test case without it being hampered by its performance at a very early stage, which may be irrelevant to its current performance, this work introduces a sliding window approach to apply different ML regression algorithms for the TCPCI problem. The hypothesis is that the use of other ML techniques, which are not based on RL, can lead to better performance. The approach helps to deal properly with the *Exploration versus Exploitation (EvE)* dilemma [10] that regards to the fact that it is necessary to prioritize the test cases with high probability to fail, but is also important to include new test cases in the prioritized set, otherwise, some test cases can never be executed given the test budget. This means that, for such problems, solutions with the best performance (exploitation) are desired, but it is also important to ensure diversity, that is, dissimilar solutions (exploration).

The approach is evaluated using the regression algorithm *Random Forest (RF)* [11] and a *Long Short Term Memory (LSTM)* deep learning neural network [12]. Furthermore, we used three time budgets and eleven systems. The results show that both, RF and LSTM, are applicable to the TCPCI problem, regarding the time to perform the prioritization and the CI cycles. Considering the *Normalized Average Percentage of Faults Detected (NAPFD)* [13] metric, RF reached the best values, or statistically equivalent to the best ones, in 23 cases (out of 33, ≈70%), while COLEMAN and LSTM presented the best values respectively, in 20 (≈61%) and 18 (≈55%) cases, and RETECS only in 8 (≈24%).

In this way, the main contributions of this paper is to introduce and evaluate an approach to the TCPCI problem. Such an approach has the following characteristics and advantages:

- It implements the sliding window method that deals properly with test case volatility;
- It is generic and can be instantiated with different algorithms. We evaluate and compare RF, a regression algorithm, and LSTM, a deep learning network, as an alternative for existing RL based approaches;
- Unlike other ML approaches, it does not require retraining of the model to perform the prioritization, that is, it is independent of a model and programming language, and does not require any code analysis;
- It is more lightweight because it considers only the result of the test case, that is, if it passed or not, and needs only the historical failure data to execute.

In addition to this, we present results evaluating the algorithm RF and an LSTM network in comparison with the state-of-the-art RL approaches: COLEMAN and RETECS. We discuss some practical implications for the usage of the algorithms and approaches evaluated. The results are available in a repository for future investigation [14].

The paper is structured as follows: Section II reviews related work on TCPCI. Section III presents the proposed approach. Section IV describes the experimental setting and research questions. Section V presents and analyses the results. Section VI discusses the main implications of the results, and limitations of the work. Section VII concludes the paper and contains some directions for future work.

II. Related Work

Marijan et al. [15]–[18] introduced different TCPCI approaches. The first one, ROCKET [15], considers the distance of the failure status from a current execution of a test case to its execution time. A posterior work of the authors [16] considers, given a test budget, other factors such as fault detection, business, performance, and technical perspectives. A tool, called TITAN [17], which is based on constraint programming, minimizes the number of test cases that cover some requirements that the original test cases cover. Then, the minimized set is prioritized using ROCKET. To address TCP in the context of Highly Configurable Systems, Marijan et al. [18] uses a learning algorithm to classify test cases considering the feature coverage and to discover redundant test cases. The priority is calculated based on its historical fault detection effectiveness and considering a test budget.

The approach of Xiao et al. [19] performs the prioritization based on test cases that failed recently. After this the test cases are ordered according to the failure history, test coverage, test size and execution time. Haghighatkhah et al. [20] tries to improve the effectiveness of the prioritization by using historical failure knowledge with a diversity measure, calculated by comparing the text from test cases. The work of Busjaeger and Xie [21] creates a fault-prediction model for the test cases by using SVM^{map}. Such a model is used in the prioritization, taking into account five attributes: test coverage of modified code, textual similarity between tests and changes, recent test-failure or fault history, and test age. But these attributes need additional information and rely on code instrumentation.

The main limitations of the works mentioned above is that they required an effort and additional cost to calculate the test coverage or other additional information. Moreover they do not address the main characteristics of CI environments - test case volatility. They are not adaptive, that is, they do not learn with past prioritizations. To overcome such limitations, more recently, *Reinforcement Learning (RL)* approaches have been proposed.

The approach, called RETECS [7] uses RL to prioritize and select test cases. RETECS considers as input the test case duration, historical failure data, and previous last execution and is guided by a reward function that learns over the cycles and can adapt to changes. The authors compared different RL variants and the ANN variant presented the best results. A similar approach, namely COLEMAN [8], also is based on reward functions but uses MAB to perform the prioritization. These last approaches present some advantages because they

properly deal with the volatility of the test cases and address the EvE dilemma [10]. To solve such a dilemma an approach needs to balance the diversity of test cases, and the quantity of new test cases and test cases that are error-prone or that comprise high fault-detection capabilities. This problem is related to the test budget, because if only error-prone test cases are considered without diversity, some test cases can never be executed. In experiments reported in the literature [8], COLEMAN outperformed RETECS, by presenting better performance, regarding NAPFD and other indicators.

But RETECS and COLEMAN present some disadvantages, they do not consider the current state of the system for the prioritization. Their performance decreases for systems with a large test case set, in which many failures are distributed over many test cases. The approach proposed in our work allows the use of traditional and deep learning techniques, as an alternative for RL. This contributes to reduce those limitations and improve performance. Moreover, our approach uses the concept of sliding window [9] to deal with test case volatility and EvE dilemma.

The application of deep learning to perform software engineering tasks has raised interest of researchers and practitioners. Mappings of the filed [22]–[24] show that most studies focus on classification tasks, and software testing is one of the main software engineering activity. We found only three papers that use deep learning for TCP. The works of Medhat et al. [25] and Raeisi et al. [26] use respectively LSTM and *Self Organizing Map (SOM)* for TCP, but the proposed approaches work in specific scenarios IoT-based components and web applications, and can not be directly applied to our context.

The approach of Xiao et al. [27] receives as input a sequence data set including the test verdicts of each execution along the test case execution history. This information is used to train the LSTM network, which produces as output the probability of the test fails in the future. A network is trained for each test case, and in the end, the test cases are ranked in descending order according to this probability. The authors also suggest a strategy for combining the LSTM approach with a deterministic one. Such a combined approach was compared against ROCKET, RETECS and a random approach, considering NAPFD.

The work of Xiao et al. is the first initiative using LSTM for TCPCI, but it has some limitations. The conducted evaluation does not include COLEMAN in the comparison the MAB-based approach that outperformed RETECS, and only considers two systems IOF-ROL and Paint Control. Experiments with COLEMAN and RETECS reported in the literature were conducted with a broader set of systems [7], [8], and point out some difficult cases for these learning approaches that were not considered. Moreover, the approach proposed does not consider the existence of a test budget. Differently, our usage of the LSTM network considers the test budget to perform the prioritization; our evaluation is conducted with 11 systems, and the results are compared with COLEMAN and RETECS.

The work of Bertolino et al. [6] presents results comparing two ML strategies in CI and evaluates the performance of different Machine Learning (ML) algorithms. The first strategy evaluated, namely Learning-to-Rank, uses supervised learning to train a model based on some test features. The model is then used to rank test sets in future commits. The problem with this strategy is that it requires retraining of the model, because many times the model may no longer be representative when the commit context changes. The second strategy evaluated, namely Ranking-to-Learn, is more suitable to the dynamic CI context. This kind of strategy is based on RL and is the focus of our work. The authors conclude that Ranking-to-Learn strategies are more robust regarding test case volatility, code changes, and number of failing tests. However the evaluation conducted by Bertolino et al. uses a different approach from ours, as well as approaches. We used another set of evaluation measures, adopted in TCP literature, and a set of larger systems, leading to new findings and insights about the learning approaches.

III. Proposed Approach

Figure 1 shows how our approach works in the CI pipeline. Firstly, the code is built from the most recent commit c. If the build is successful, then the set of tests in the commit to be analyzed, Tc, is integrated into the historical data set that contains all the test cases from previous commits. This set is used by the ML algorithm to fault-proneness prediction. The next step filters the test cases $t \in Tc$ from the output, sorts them in descending order, turning them into the prioritized set $T'c$, which is executed by the CI environment. Finally, $T'c$ is evaluated and its results are saved as historical data set for the next iteration of the environment. Only a simple attribute regarding the test cases results is saved: whether they failed or not in a commit c.

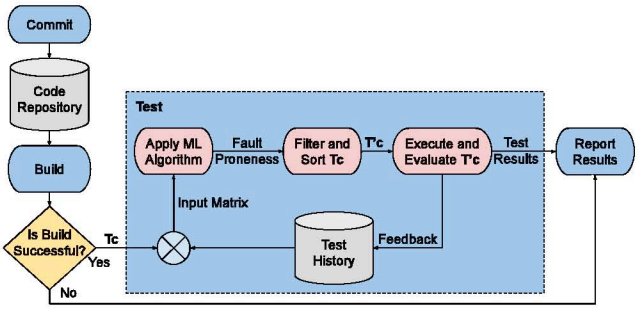


Fig. 1: Overview of our approach applied to the CI environment.

Figure 2 represents in more detail how the approach handles the data. The ML model is used to predict the fault-proneness of the existing test cases in the current commit using historical test data represented in an input matrix. In such a matrix 1 is used to represent a failure and 0 otherwise. As a result, a vector with the fault proneness of each test case is produced. Then the vector is sorted and finally filtered considering the test cases $t \in Tc$, generating a vector with only the test cases that are going to be executed. The algorithm treats test cases

that were renamed or changed as new, as for the past test case, its relevance will drop after it passes the SW threshold.

To deal with the volatility of test cases we employ the Sliding Window SW method [9]. This concept allows the ML algorithm to use a window of size W, containing data of the previous W commits as the source of information, moving the window to provide new historical data when necessary. In this way, the analysis is not limited to a fixed set of test cases. Figure 3 shows how the SW method works using W equals to 2. First, the commit c_1 is analyzed. For which there is no historical data, thus the SW is empty. In the analysis of commit c_2, the SW is filled with information from the previously evaluated commit c_1. The first part of the figure represents the evaluation of the commit c_3, in which the SW is filled with the values of c_1 and c_2. For the analysis of commit c_4, c_1 is dropped from the SW, making room for the set formed by c_2 and c_3.

It is important to note that the SW method used adopts only the most recent data, that is, only the W most recent commits to be used as input for the ML algorithms. In our implementation, the ML algorithm is used only after at least 3 commits, when a historical test database is available with enough data to make the prediction, otherwise a random prioritization is used.

To evaluate our approach and the SW method, our implementation uses two learning algorithms. The first one is *Random Forest* (RF) [11]. As represented in (Figure 4a) RF is an ensemble of decision trees. The decision trees make their predictions independently and the output of the algorithm is obtained by averaging their results. This algorithm was chosen because it presented the best results according to different criteria evaluated for RL strategies in the work of Bertolino et al. [6].

We also implemented an LSTM network [12]. The LSTM network consists of multiple LSTM cells as recurrent units. Each cell has its own state. The LSTM structure includes some key components, which are the gates [28]. The forget gate is responsible for controlling which information from previous layers is relevant enough to be transferred to subsequent layers. The input gate and the cell gate are responsible for evaluating new information present in the LSTM inputs, integrating relevant information into the network. They work with the hidden state information from the previous LSTM cell and with the new input from the current state. A cell update action modifies the cell state value, which contains the value resulting from operations already performed without using the output gate. The result of this operation is used in the calculation of the output gate. The output gate is responsible for processing and controlling the output from the network and for propagating information to the next layer.

Figure 4b represents the developed LSTM model that is composed of several layers. Firstly, the input is given to the algorithm, then this input goes to the input layer, and next it is processed by the first $tanh$ activation layer followed by a dropout layer; then a second $tanh$ activation layer leads to a dense layer, and finally to the output layer that produces the fault proneness predictions.

We chose LSTM due to its capacity to remember things. It is capable to simulate a memory and has presented noticeably promising results in different domains and for time-series prediction [28]. We then conjecture that it can improve performance and can be very effective for TCPCI. In addition to this, it has not been evaluated in the work of Bertolino et al., and has not been compared against the MAB based approach.

IV. Experimental Setting

The main hypothesis of this work is that our sliding window approach using the LSTM neural network and the RF algorithm can be used to address the TCPCI problem in a very cost-effective way, improving performance regarding related work. In this section we present the setup of the experiment we conducted to evaluate such a hypothesis.

A. Research Questions

According to our goals, we formulated the following research questions:

RQ1: *Is the approach using LSTM and RF applicable in the CI environments?* This question investigates the applicability of the approach with both algorithms. We consider the time to perform the CI cycles and the time both algorithms spend to perform the prioritization.

RQ2: *How are the performance of LSTM and RF compared against the approaches* `COLEMAN` *and* `RETECS`*?* This question compares both algorithms with the RL approaches `COLEMAN` and `RETECS` that can be considered the state-of-the-art TCPCI approaches. They are evaluated according to some common measures used in the regression testing literature (Section IV-C).

B. Used Systems

To allow comparison with existing approaches, we adopted a set of systems already used in the literature [7], [8], [29]. We used `COLEMAN` and `RETECS` results reported in the work of Prado Lima and Vergilio [8]. `COLEMAN` was executed with the policy FRRMAB, and `RETECS` with a neural network. We adopted the same budgets used by the authors: 10%, 50% and 80%.

The target systems are detailed in Table I that contains: the system name, the period of build logs analyzed, the total of builds identified, and in parentheses, the number of builds included in the analysis. Build logs with some problems were discarded, e.g., extracting information (non-valid build log), and those the test cases did not execute. The fourth column shows the total of failures found; in parentheses, the number of builds in which at least one test failed. The fifth column shows the number of unique test cases identified from build logs; in parenthesis, the range of test cases executed in the builds. The sixth and seventh columns present the average duration and standard deviation in minutes of the CI Cycles (commits), and the interval between them.

More details about these systems are available in Prado Lima and Vergilio replication package [30]. This package

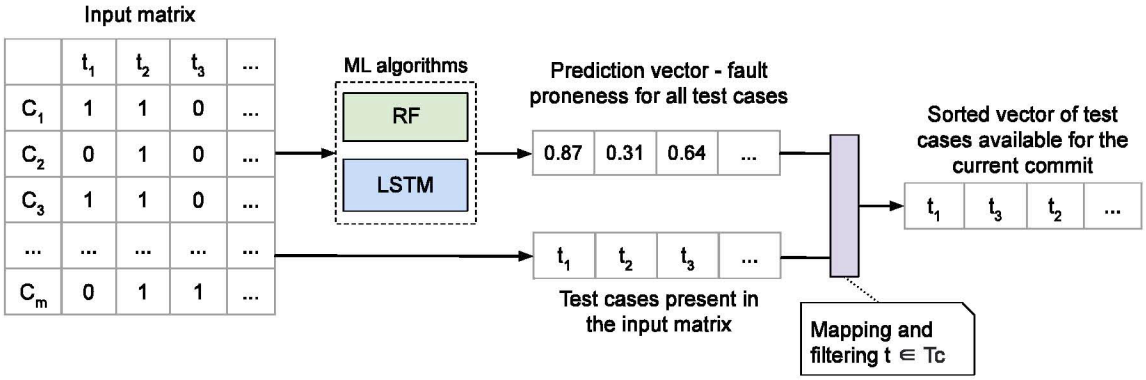


Fig. 2: Approach Overview containing input matrix, ML algorithms, and how the output is produced.

TABLE I: Used Systems.

Name	Period	Builds	Failures	Test Cases	Duration (min)	Interval (min)
Druid	2016/04/24-2016/11/08	286 (168)	270 (71)	2391 (1778-1910)	4.97 ± 10.66	384.76 ± 468.86
Fastjson	2016/04/15-2018/12/04	2710 (2371)	940 (323)	2416 (900-2102)	1.97 ± 0.89	233.22 ± 401.26
Deeplearning4j	2014/02/22-2016/01/01	3410 (483)	777 (323)	117 (1-52)	12.33 ± 14.91	306.05 ± 442.55
DSpace	2013/10/16-2019/01/08	6309 (5673)	13413 (387)	211 (16-136)	11.78 ± 7.03	291.29 ± 411.19
Guava	2014/11/06-2018/12/02	2011 (1689)	7659 (112)	568 (308-512)	62.53 ± 80.31	435.55 ± 464.52
IOF/ROL	2015/02/13-2016/10/25	2392 (2392)	9289 (1627)	1941 (1-707)	1537.27 ± 2018.73	1324.36 ± 291.78
LexisNexis	2018/09/27-2018/11/15	54 (54)	21189 (54)	2662 (2007-2377)	0.8668 ± 0.808	900.367 ± 305.125
OkHttp	2013/03/26-2018/05/30	9919 (6215)	9586 (1408)	289 (2-75)	7.64 ± 5.64	220.17 ± 405.93
Paint Control	2016/01/12-2016/12/20	20711 (20711)	4956 (1980)	1980 (1-74)	424.46 ± 275.90	1417.86 ± 144.97
Retrofit	2013/02/17-2018/11/26	3719 (2711)	611 (125)	206 (5-75)	2.40 ± 1.60	270.86 ± 449.41
ZXing	2014/01/17-2017/04/16	961 (605)	68 (11)	124 (81-123)	13.14 ± 12.37	411.10 ± 465.53

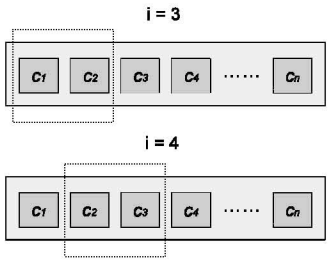


Fig. 3: Example of a Sliding Window with size 2.

contains some figures illustrating number of failures per cycle for each SUT, which allows observing test case volatility.

C. Evaluation Measures

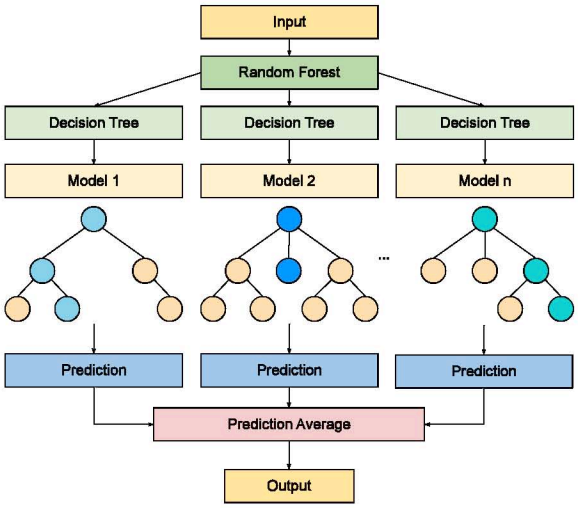
To evaluate the approaches we adopted measures commonly used in the regression testing area: *Normalized Average Percentage of Faults Detected (NAPFD)*, and *Average Percentage of Faults Detected with cost consideration (APFDc)*, which are extensions of the *Average Percentage of Faults Detected (APFD)* [31]. APFD indicates how quickly a set of prioritized test cases (T') can detect the faults present in the application being tested, and its value is calculated from the weighted average of the percentage of detected faults. APFD values range from zero to one. Higher values indicate that the faults are detected faster using fewer test cases. NAPFD considers the ratio between detected and detectable faults within T. This metric is adequate for prioritization of test cases when not all of them are executed, and some faults can be undetected [13].

$$NAPFD(T'_t) = p - \frac{\sum_1^n rank(T'_{t_i})}{m \times n} + \frac{p}{2n} \quad (1)$$

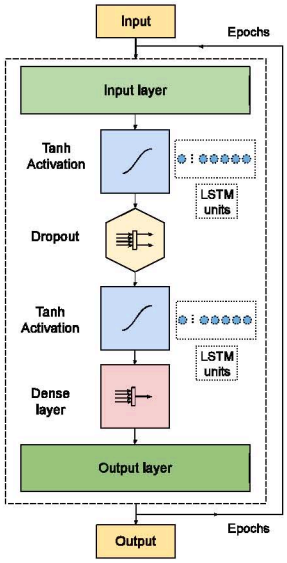
where m is the number of faults detected by all test cases; $rank(T'_i)$ is the position of T'_i in T', if T'_i did not reveal a fault we set $T'_i = 0$; n is the number of tests cases in T'; and p is the number of faults detected by T' divided by m. The last part of the equation represents the full area under the curve when the percentage of faults found is plotted on the y-axis and the percentage of the run test cases is on the x-axis. NAPFD is equal to APFD metric if all faults are detected.

APFDc assumes that the test cases do not have the same cost, that is, some may be more costly to execute than others, maybe due to the fault severity or running time. The test cases are usually prioritized until a maximum cost is reached, which is feasible to execute. Furthermore, if both fault severity and test case costs are identical, APFDc can be used to compute the APFD value. In this work, we consider that all faults have same severity.

$$APFDc(T'_t) = \frac{\sum_{i=1}^m (\sum_{j=TF_i}^n c_j - 0.5 c_{TF_i})}{\sum_{j=1}^n c_j \times m} \quad (2)$$



(a) Overview of Random Forest algorithm.



(b) Overview of the LSTM network.

Fig. 4: Overview of the used algorithms.

where c_i is the cost of a test case T_i, and TF_i is the first test case from T' that reveals fault i.

Furthermore, we used the metric *Normalized Time Reduction (NTR)* [8]. This metric is used to observe the difference between time spent until the first test case fails r_t and the total time spent to execute all tests $\hat{r}_t$ as defined by the Equation 3. In the calculation, only the commits that failed, CI^{fail}, are considered. In this way, we can evaluate the capability of an algorithm to reduce the time spent in a CI cycle.

$$NTR(\mathcal{A}) = \frac{\sum_{t=1}^{CI^{fail}} (\hat{r}_t - r_t)}{\sum_{t=1}^{CI^{fail}} (\hat{r}_t)} \quad (3)$$

The last measure used is the Prioritization Time (PT), which calculates the time spent (in seconds) by an algorithm to perform the prioritization. This measure helps to observe whether the evaluated algorithms take a reasonable time to prioritize, if the prioritization time is too high, the use of the algorithm may be unfeasible for real scenarios.

We applied Kruskal-Wallis [32], Mann-Whitney [33], and Friedman [34] statistical tests with a confidence level of 95%. We used Kruskal-Wallis to evaluate the performance of the approaches in each system. Due to lack of time we adopted 10 independent runs. For comparison reasons, we used the results of only the first 10 executions of `COLEMAN` and `RETECS` [8]. We applied Mann-Whitney to evaluate a group of performances in the same system or for post-hoc analysis. We used Friedman to evaluate the approach behavior across different systems. To this end, each system becomes a dependent variable, in which we apply multiple approaches.

Additionally, to calculate the effect size magnitude of the difference between groups, we used the Vargha and Delaney's $\hat{A}_{12}$ [35] metric. This measure ranges from 0 to 1 and defines the probability of a value, taken randomly from the first sample, is higher than a value taken randomly from the second sample. A $Negligible$ magnitude ($\hat{A}_{12} < 0.56$) represents a very small difference among the values and usually does not yield a statistical difference. The $Small$ ($0.56 \leq \hat{A}_{12} < 0.64$) and $Medium$ ($0.64 \leq \hat{A}_{12} < 0.71$) magnitudes represent small and medium differences among the values, and may or not yield statistical differences. Finally, a $Large$ magnitude ($0.71 \leq \hat{A}_{12}$) represents a significantly large difference that usually can be seen in the numbers without much effort.

D. Implementation and Configuration

To implement the LSTM network we used the cuDNN library [36], which through Tensorflow [37] and Keras [38] frameworks, allows the network to use GPUs. Some parameters were set with default values as described on the Keras guide[1] for CuDNNLSTM development. They are: activation function: $tanh$; recurring activation function: sigmoid; recurring dropout: zero; unroll: False; use of bias: True; If the input data contains a mask, they must be strictly filled in on the right; and fast execution is enabled in the outermost context.

To configure the network, we conducted tests using the `Deeplearning4j` system, by training different configurations. We first defined the model as sequential, and conducted experiments varying the number of epochs in the interval [10, 100], and the number of neurons in the interval [25, 256]. After these tests, we adopted 100 neurons for each layer, and 100 epochs for the model. With these parameters selected, we started experimenting with different layer configurations. We tested a total of 5 configurations using in all of them the $tanh$ activation layer. The configurations performed alternated

[1]https://keras.io/api/layers/recurrent_layers/lstm

in addition of activation layers *tanh* followed by a layer of *dropout* with a value of 0.2, considering as baseline a configuration with 3 layers containing firstly a *tanh* activation layer, followed by a *dropout* and another *tanh* activation layer. Following this baseline, the tests were carried out using the settings for 3, 5, 7, 9 and 11 layers. In the end, we selected the baseline layer configuration (3 layers) due to its lower prioritization and comparable accuracy to other configurations. For all the tests performed, we used optimizer Adam and the loss as Mean Square Error (MSE).

The experiments were performed on 2 platforms which provided the use of GPUs: Google Colab Pro[2] and Fed4Fire[3] (model: GV100GL [Tesla V100 PCIe 32GB]). We used results from 10 independent runs for each dataset due to the high computational cost and the large amount of time required to train the network. We adopted the size W equals 100.

To implement the RF regression algorithm, we used the Scikit Learn [39] which uses the average of the predictions of the generated trees. The experiments with RF were conducted in the GENI [40] (Global Environment for Network Innovations) platform with the following configuration: Intel(R) Xeon(R) E5-2640 v3 with 2.60 GHz CPU, 94GB RAM, running Linux Ubuntu 18.04.1 LTS. To set the parameters, number of estimators (number of trees generated) and W, we selected the systems `DeepLearning4J` and `FastJson`, performing 10 independent runs.

We tested variations of 100, 250 and 500 estimators and a variation of 0, 1, 4, 100 for the W, in which the size 0 represents a cumulative approach, where all the historical test data was used. These sizes were selected based on the literature to allow comparison against approaches where: $W = 1$ (no historical test data is available); where $W = 4$ (used by Bertolino et al. [6]); and where $W = 100$ (adopted by Prado Lima and Vergilio [8]). Regarding the number of estimators, 500 is considered the default value in the literature [41]; 100 estimated by Hartshorn [42] to be a great start in the tuning phase; and 250 to score an intermediate value.

During the tuning phase, we noticed that $W = 0$ was impracticable, as in some systems with a high number of test cases and commits, the prioritization time surpassed 60 seconds. The best overall performance was achieved using $W = 4$; A value for W equals to 100 had a good performance only for `FastJson`, W equals to 1 performed well only for `DeepLearning4J`. Then $W = 4$ was chosen. We noticed that 500 estimators had the best overall performance, scoring greatly in both systems, specially with W equals to 4. We used the results of 10 executions of RF to compare with LSTM, as well as `COLEMAN` and `RETECS`.

V. Results and Analysis

In this section, we present and analyze the results, aiming to answer our research questions.

[2]https://colab.research.google.com

[3]https://portal.fed4fire.eu

A. RQ1: Applicability considering CI cycles

To answer RQ1, we compare the build time of each dataset, presented in Table I, with the execution time of RF and LSTM, presented in Table II. In such a table, the values were obtained from the 10 independent runs with time budgets of 10%, 50% and 80%. The cells in bold contain the best values. With this information we can conclude whether they can solve the TCPCI problem in a way that the time to run the tests is justifiable to use our approach. We can observe that both algorithms are applicable to all systems; the time spent by them does not overload the cycle execution time. Both have a lower execution time than the build time and interval between builds of the systems.

The prioritization time for the analyzed algorithms is present in Table II. This table shows that both proposed algorithms take only few seconds to execute. In the worst case, considering the standard deviation, they take around 7 seconds (RF for `Lexisnexis` and LSTM for `Druid`). Except for `Lexisnexis`, RF presents the best PT values. LSTM takes around 4 times longer to execute than RF. However, by analyzing the NTR values (Table III), we can see that this fact does not impact the time spent in the CI cycle. The NTR values represent how much the build execution time improves due to the early fault prediction as the effect of the prioritization of test cases, and for the duration of the dataset builds. The performance of both algorithms is similar. Considering the 33 cases (11 systems x 3 budgets), LSTM presents better values than RF in 19 cases (58%), and RF presents better values than LSTM in 14 (42%).

TABLE II: Prioritization Time (in seconds) and standard deviation for `COLEMAN`, `RETECS`, LSTM and RF.

SUT	NTR			
	RETECS	COLEMAN	LSTM	RF
	Time Budget: 10%			
DSpace	0.0310 ± 0.0020	**0.0300 ± 0.0020**	4.8780 ± 0.4940	1.1090 ± 0.0180
Druid	0.3220 ± 0.0290	**0.2320 ± 0.0040**	7.1360 ± 0.0160	1.3110 ± 0.0670
Fastjson	**0.2050 ± 0.0180**	0.4510 ± 0.2200	5.2120 ± 0.5710	1.4340 ± 0.0330
Deeplearning4j	**0.0170 ± 0.0030**	0.0270 ± 0.0000	3.7390 ± 0.0400	1.0770 ± 0.0440
Guava	**0.0400 ± 0.0040**	0.0680 ± 0.0140	4.9530 ± 0.0640	1.1530 ± 0.0220
IOF/ROL	0.1900 ± 0.0290	**0.0280 ± 0.0000**	4.9040 ± 0.6070	1.1240 ± 0.0090
Lexisnexis	0.5990 ± 0.0570	**0.3300 ± 0.0060**	4.5930 ± 0.1230	6.8370 ± 0.1420
Paint Control	**0.0020 ± 0.0000**	0.0260 ± 0.0000	4.6140 ± 0.0070	1.1000 ± 0.0070
OkHttp	**0.0220 ± 0.0020**	0.0290 ± 0.0010	4.6090 ± 0.0070	1.1090 ± 0.0130
Retrofit	**0.0100 ± 0.0020**	0.0280 ± 0.0000	4.5800 ± 0.0080	1.1020 ± 0.0100
ZXing	**0.0150 ± 0.0010**	0.0340 ± 0.0000	4.6010 ± 0.5700	1.1020 ± 0.0280
	Time Budget: 50%			
DSpace	0.0510 ± 0.0020	**0.0290 ± 0.0010**	4.8820 ± 0.4820	1.1260 ± 0.0190
Druid	0.3810 ± 0.0540	**0.2380 ± 0.0020**	7.1390 ± 0.0100	1.3330 ± 0.0400
Fastjson	**0.2440 ± 0.0310**	0.4420 ± 0.2030	5.2110 ± 0.5510	1.4380 ± 0.0310
Deeplearning4j	**0.0180 ± 0.0030**	0.0270 ± 0.0000	3.7530 ± 0.0560	1.1060 ± 0.0170
Guava	**0.0410 ± 0.0040**	0.0650 ± 0.0130	4.9920 ± 0.1810	1.1610 ± 0.0150
IOF/ROL	0.4770 ± 0.1080	**0.0280 ± 0.0000**	4.8970 ± 0.6010	1.1180 ± 0.0170
Lexisnexis	1.6550 ± 0.2870	**0.3710 ± 0.0020**	4.5310 ± 0.0940	6.5440 ± 0.6570
Paint Control	**0.0030 ± 0.0000**	0.0260 ± 0.0000	4.6140 ± 0.0070	1.1110 ± 0.0100
OkHttp	0.0360 ± 0.0020	**0.0280 ± 0.0000**	4.6080 ± 0.0070	1.1250 ± 0.0140
Retrofit	**0.0170 ± 0.0030**	0.0280 ± 0.0000	4.5810 ± 0.0060	1.1040 ± 0.0190
ZXing	**0.0240 ± 0.0080**	0.0340 ± 0.0000	4.5850 ± 0.5510	1.1010 ± 0.0120
	Time Budget: 80%			
DSpace	0.0550 ± 0.0010	**0.0290 ± 0.0020**	5.0210 ± 0.6560	1.1280 ± 0.0190
Druid	0.2740 ± 0	**0.2410 ± 0.0030**	7.1390 ± 0.0100	1.3330 ± 0.0530
Fastjson	**0.2370 ± 0.0310**	0.4450 ± 0.2050	5.0500 ± 0.5550	1.3960 ± 0.0280
Deeplearning4j	**0.0200 ± 0.0080**	0.0270 ± 0.0000	3.7530 ± 0.0640	1.0950 ± 0.0260
Guava	**0.0400 ± 0.0030**	0.0650 ± 0.0130	4.9720 ± 0.0830	1.1450 ± 0.0310
IOF/ROL	0.7160 ± 0.0830	**0.0280 ± 0.0000**	4.8280 ± 0.5860	1.1200 ± 0.0090
Lexisnexis	5.1120 ± 0.9690	**0.4070 ± 0.0040**	4.5280 ± 0.0910	6.6670 ± 0.3180
Paint Control	**0.0030 ± 0.0010**	0.0260 ± 0.0000	4.6140 ± 0.0070	1.1120 ± 0.0110
OkHttp	0.0410 ± 0.0010	**0.0280 ± 0.0000**	4.6080 ± 0.0070	1.1260 ± 0.0160
Retrofit	**0.0200 ± 0.0030**	0.0280 ± 0.0000	4.5820 ± 0.0040	1.1100 ± 0.0060
ZXing	**0.0300 ± 0.0180**	0.0340 ± 0.0000	4.6300 ± 0.5930	1.1030 ± 0.0170

TABLE III: Mean and standard deviation NTR values: COLEMAN, RETECS, LSTM and RF.

SUT	NTR			
	RETECS	COLEMAN	LSTM	RF
TIME BUDGET: 10%				
DSpace	0.0102 ± 0.001	**0.0251 ± 0.004**	0.0188 ± 0.001	0.0160 ± 0.000
Druid	0.1973 ± 0.107	0.1599 ± 0.100	0.3772 ± 0.007	**0.3841 ± 0.002**
Fastjson	0.0213 ± 0.008	0.0604 ± 0.024	0.1027 ± 0.002	**0.1077 ± 0.000**
Deeplearning4j	**0.5432 ± 0.019**	0.4663 ± 0.000	0.4624 ± 0.011	0.4748 ± 0.000
Guava	0.0388 ± 0.004	0.0330 ± 0.002	**0.0524 ± 0.002**	0.0509 ± 0.000
IOF/ROL	0.5634 ± 0.012	**0.5710 ± 0.003**	0.5218 ± 0.152	0.4751 ± 0.000
Lexisnexis	0.9861 ± 0.010	0.9961 ± 0.000	**0.9988 ± 0.000**	0.9981 ± 0.000
Paint Control	**0.1139 ± 0.000**	0.1131 ± 0.000	0.0967 ± 0.001	0.0955 ± 0.000
OkHttp	0.0541 ± 0.007	0.0702 ± 0.000	0.0675 ± 0.002	**0.0776 ± 0.000**
Retrofit	0.0074 ± 0.001	0.0073 ± 0.000	0.0074 ± 0.000	**0.0076 ± 0.000**
ZXing	**0.0123 ± 0.000**	0.0037 ± 0.000	0.0106 ± 0.002	0.0079 ± 0.000
TIME BUDGET: 50%				
DSpace	0.0219 ± 0.002	**0.0499 ± 0.006**	0.0366 ± 0.002	0.0211 ± 0.000
Druid	0.0982 ± 0.020	**0.4212 ± 0.008**	0.3906 ± 0.005	0.3922 ± 0.002
Fastjson	0.0666 ± 0.026	0.0768 ± 0.028	**0.1171 ± 0.001**	0.1155 ± 0.000
Deeplearning4j	**0.5462 ± 0.010**	0.4624 ± 0.000	0.4795 ± 0.012	0.4863 ± 0.000
Guava	0.6760 ± 0.018	**0.7185 ± 0.003**	0.6658 ± 0.004	0.6641 ± 0.000
IOF/ROL	0.9908 ± 0.006	0.9961 ± 0.000	**0.9988 ± 0.000**	0.9980 ± 0.000
Lexisnexis	0.0410 ± 0.005	0.0419 ± 0.004	**0.0576 ± 0.001**	0.0568 ± 0.000
Paint Control	0.1138 ± 0.001	**0.1223 ± 0.000**	0.0967 ± 0.001	0.0956 ± 0.000
OkHttp	0.1052 ± 0.003	**0.1486 ± 0.000**	0.0712 ± 0.002	0.0792 ± 0.000
Retrofit	0.0138 ± 0.000	**0.0172 ± 0.000**	0.0139 ± 0.001	0.0140 ± 0.000
ZXing	**0.0199 ± 0.001**	0.0110 ± 0.000	0.0171 ± 0.001	0.0184 ± 0.000
TIME BUDGET: 80%				
DSpace	0.0266 ± 0.001	**0.0526 ± 0.005**	0.0400 ± 0.001	0.0238 ± 0.000
Druid	0.1313 ± 0.114	**0.4289 ± 0.005**	0.3955 ± 0.010	0.4014 ± 0.002
Fastjson	0.0553 ± 0.014	0.0902 ± 0.026	**0.1207 ± 0.001**	0.1190 ± 0.000
Deeplearning4j	**0.5491 ± 0.006**	0.4047 ± 0.000	0.4828 ± 0.007	0.4880 ± 0.000
Guava	0.6891 ± 0.006	**0.7329 ± 0.002**	0.6842 ± 0.005	0.6830 ± 0.000
IOF/ROL	0.9880 ± 0.006	0.9961 ± 0.000	**0.9988 ± 0.000**	0.9981 ± 0.000
Lexisnexis	0.0336 ± 0.011	0.0557 ± 0.011	**0.0590 ± 0.001**	0.0590 ± 0.000
Paint Control	0.1158 ± 0.001	**0.1203 ± 0.000**	0.0968 ± 0.001	0.0956 ± 0.000
OkHttp	0.1157 ± 0.002	**0.1551 ± 0.000**	0.0721 ± 0.002	0.0800 ± 0.000
Retrofit	0.0145 ± 0.001	**0.0179 ± 0.000**	0.0145 ± 0.000	0.0147 ± 0.000
ZXing	0.0187 ± 0.003	**0.0227 ± 0.000**	0.0175 ± 0.001	0.0184 ± 0.000

Answer to RQ1: we can conclude that LSTM and RF are applicable to solving the TCPCI problem. They require, in most cases, a few seconds to execute, and the prioritization performed is feasible in comparison with the build time. But LSTM takes almost 4 times longer than RF to perform the prioritization, but this does not mean a reduced improvement in the time to execute the CI cycle.

B. RQ2: Comparing with state-of-the-art RL approaches

To answer RQ2 we evaluate the values for NTR (Table III), NAPFD and APFDc (Table IV). As mentioned before, these values (averages ± standard deviation) were obtained from 10 executions for the three test budgets, due to time constraints training the LSTM model. Due to different execution environments, the prioritization time could not be used in our analysis, but they are present on Table II as reference.

In Table IV, values highlighted in bold with a "★" symbol denotes that the algorithm achieved the best score within the group. In cases where the results are statistically equivalent to the best one, the cell is highlighted in gray. If there is no statistical difference, both algorithms are counted as the best in our analysis. A "▼" indicates that the effect size was negligible in relation to the best value, while "▽" denotes a small magnitude, "△" a medium magnitude, and "▲" a large magnitude [8]. The effect size was performed during the post-hoc tests, that is, when there is a statistical difference. We can see in both tables that most values have the large magnitude (symbol "▲").

Analyzing the NAPFD values (Table IV), both RF and LSTM have good performance in the budget of 10%, reaching, the best values or statistically equivalent, respectively in 9 and 7 systems (out of 11), outperforming COLEMAN and RETECS, which are the best, respectively, in 5 and 3 systems.

When analyzing the time budget of 50%, we notice that RF presents the best values in 8 systems, COLEMAN and LSTM have the same score, reaching the best values in 6 systems, and RETECS only in 2. These results show that for mid and low budgets, considering only NAPFD values, the RF algorithm outperforms the others. On the other hand, for the budget of 80%, COLEMAN has the best performance in 9 systems, while RF in 6, LSTM in 5, and RETECS in 3. Considering NAPFD, all systems and budgets, RF reached the best values, or statistically equivalent to the best ones, in 23 cases (out of 33, ≈70%), while COLEMAN and LSTM presented the best values respectively, in 20 (≈61%) and 18 (≈55%) cases, and RETECS only in 8 (≈24%).

Regarding APFDc values (Table IV), and considering the budget of 10%, RF and LSTM have the best performance in 7 systems, followed by RETECS in 6, and COLEMAN in 4. RF also presented the best performance for the budget of 50% in 7 systems, followed by RETECS and COLEMAN that were the best in 6 systems. LSTM was the best in 5. This shows that RETECS performs better when the test cases cost is considered. Again, for the budget of 80%, COLEMAN achieves the best score in 9 systems, followed by RF in 5, LSTM and RETECS in 4. For APFDc, all systems and budgets, RF and COLEMAN reached the best values, or statistically equivalent to the best ones, in 19 cases (out of 33, ≈58%), while RETECS in 18 (≈55%) and LSTM in 16 (≈48%). The algorithms have a similar overall performance.

Finally, analyzing the NTR values (Table III), we can observe that RF has the best values in 4 systems for the 10% time budget and the worst performance for the budgets of 50% and 80% (0 systems). COLEMAN reached the best values for the budgets of 50% and 80%, respectively, in 6 and 7 systems. But presents a low performance in the budget of 10% (2 cases).

Answer to RQ2: We can conclude that RF outperforms the other algorithms for more restrictive and mid budgets, presenting the best NAPFD and APFDc values. COLEMAN outperforms the other algorithms for the less restrictive budget. The LSTM network has good performance for the budget of 10%, and RETECS when APFDc is considered, for low and mid budgets.

VI. DISCUSSION AND LIMITATIONS

In this section we discuss the implication of the results obtained and possible limitations of the approaches and adopted algorithms, as well as threats to the validity.

TABLE IV: Mean and standard deviation NAPFD and APFDc values for: RETECS, COLEMAN, LSTM and RF.

SUT	NAPFD				APFDc			
	RETECS	COLEMAN	LSTM	RF	RETECS	COLEMAN	LSTM	RF
TIME BUDGET: 10%								
DSpace	0.9408 ± 0.0000 ▲	**0.9496 ± 0.0040** ★	0.9473 ± 0.0010 ▼	0.9485 ± 0.0000 ▼	0.9456 ± 0.0000 ▲	**0.9513 ± 0.0040** ★	0.9469 ± 0.0010 ▲	0.9473 ± 0.0000 ▼
Druid	0.6919 ± 0.1080 ▲	0.6718 ± 0.0540 ▲	0.9161 ± 0.0050 ▲	**0.9284 ± 0.0020** ★	0.8089 ± 0.0710 ▲	0.6754 ± 0.0560 ▲	0.9170 ± 0.0040 ▲	**0.9307 ± 0.0010** ★
Fastjson	0.8716 ± 0.0050 ▲	0.9014 ± 0.0220 ▲	0.9491 ± 0.0010 ▲	**0.9572 ± 0.0000** ★	0.8806 ± 0.0040 ▲	0.9009 ± 0.0220 ▲	0.9493 ± 0.0010 ▲	**0.9572 ± 0.0000** ★
Deeplearning4j	0.6729 ± 0.0150 ▲	0.7716 ± 0.0000 ▲	0.7652 ± 0.0040 ▲	**0.7771 ± 0.0000** ★	**0.8109 ± 0.0100** ★	0.7774 ± 0.0010 ▲	0.7657 ± 0.0030 ▲	0.7784 ± 0.0000 ▲
Guava	0.9578 ± 0.0040 ▲	0.9584 ± 0.0010 ▲	**0.9800 ± 0.0010** ★	0.9796 ± 0.0000 ▽	0.9770 ± 0.0030 ▲	0.9580 ± 0.0010 ▲	**0.9803 ± 0.0010** ★	0.9798 ± 0.0000 ▽
IOF/ROL	**0.3779 ± 0.0030** ★	0.3671 ± 0.0010 ▲	0.3557 ± 0.0010 ▲	0.3577 ± 0.0000 ▲	**0.3820 ± 0.0030** ★	0.3701 ± 0.0010 ▲	0.3558 ± 0.0010 ▲	0.3576 ± 0.0000 ▲
Lexisnexis	0.0938 ± 0.0600 ▲	0.1437 ± 0.0010 ▲	0.3457 ± 0.0040 ▲	**0.3686 ± 0.0000** ★	0.0959 ± 0.0580 ▲	0.1431 ± 0.0010 ▲	0.3424 ± 0.0040 ▲	**0.3650 ± 0.0000** ★
Paint Control	**0.9077 ± 0.0000** ★	0.9076 ± 0.0000 ▲	0.9074 ± 0.0000 ▲	0.9074 ± 0.0000 ▲	**0.9081 ± 0.0000** ★	0.9080 ± 0.0000 ▲	0.9074 ± 0.0000 ▲	0.9074 ± 0.0000 ▲
OkHttp	0.8072 ± 0.0060 ▲	0.8407 ± 0.0000 ▲	0.8356 ± 0.0010 ▲	**0.8513 ± 0.0000** ★	0.8268 ± 0.0060 ▲	0.8425 ± 0.0000 ▲	0.8347 ± 0.0010 ▲	**0.8482 ± 0.0000** ★
Retrofit	0.9619 ± 0.0020 ▲	0.9642 ± 0.0000 ▲	0.9656 ± 0.0000 ▲	**0.9659 ± 0.0000** ★	**0.9672 ± 0.001** ★	0.9646 ± 0.000 ▲	**0.9655 ± 0.001** ★	0.9648 ± 0.000 △
ZXing	0.9855 ± 0.0000 ▽	0.9828 ± 0.0000 ▲	**0.9864 ± 0.0020** ★	0.9860 ± 0.0000 ▽	**0.9893 ± 0.0000** ★	0.9835 ± 0.0000 ▲	0.9860 ± 0.0010 ▲	0.9855 ± 0.0000 ▲
TIME BUDGET: 50%								
DSpace	0.9487 ± 0.0010 ▲	**0.9766 ± 0.0080** ★	0.9665 ± 0.0010 ▲	0.9495 ± 0.0000 ▲	0.9615 ± 0.0010 ▲	**0.9767 ± 0.0090** ★	0.9666 ± 0.0010 ▲	0.9485 ± 0.0000 ▲
Druid	0.6052 ± 0.0080 ▲	**0.9691 ± 0.0080** ★	0.9359 ± 0.0060 ▲	0.9417 ± 0.0020 ▲	0.7426 ± 0.0360 ▲	**0.9770 ± 0.0090** ★	0.9370 ± 0.0060 ▲	0.9412 ± 0.0010 ▲
Fastjson	0.8936 ± 0.0170 ▲	0.9171 ± 0.0340 ▲	**0.9636 ± 0.0010** ★	0.9632 ± 0.0000 ▲	0.9423 ± 0.0210 ▲	0.9191 ± 0.0330 ▲	0.9644 ± 0.0010 ▲	**0.9654 ± 0.0000** ★
Deeplearning4j	0.6559 ± 0.0160 ▲	0.8200 ± 0.0000 ▲	0.8403 ± 0.0050 ▲	**0.8501 ± 0.0000** ★	**0.8432 ± 0.0100** ★	0.8135 ± 0.0010 ▲	0.8295 ± 0.0060 ▲	0.8372 ± 0.0000 △
Guava	0.4995 ± 0.0070 ▲	**0.5192 ± 0.0020** ★	0.4794 ± 0.0030 ▲	0.4816 ± 0.0000 ▲	0.5016 ± 0.0070 ▲	**0.5226 ± 0.0020** ★	0.4797 ± 0.0030 ▲	0.4815 ± 0.0000 ▲
IOF/ROL	0.2706 ± 0.1340 ▲	0.5617 ± 0.0080 ▲	0.6251 ± 0.0020 ▲	**0.6487 ± 0.0000** ★	0.2807 ± 0.1250 ▲	0.5487 ± 0.0080 ▲	0.6105 ± 0.0020 ▲	**0.6328 ± 0.0000** ★
Lexisnexis	0.9595 ± 0.0040 ▲	0.9668 ± 0.0050 ▲	**0.9849 ± 0.0010** ★	0.9834 ± 0.0000 ▲	0.9828 ± 0.0020 ▲	0.9665 ± 0.0060 ▲	**0.9851 ± 0.0010** ★	0.9840 ± 0.0000 ▲
Paint Control	0.9138 ± 0.0000 ▲	**0.9150 ± 0.0000** ★	0.9074 ± 0.0000 ▲	0.9074 ± 0.0000 ▲	0.9139 ± 0.0000 ▲	**0.9162 ± 0.0000** ★	0.9075 ± 0.0000 ▲	0.9074 ± 0.0000 ▲
OkHttp	0.8433 ± 0.0030 ▲	**0.9316 ± 0.0000** ★	0.8395 ± 0.0020 ▲	0.8534 ± 0.0000 ▲	0.8863 ± 0.0020 ▲	**0.9246 ± 0.0000** ★	0.8386 ± 0.0020 ▲	0.8500 ± 0.0000 ▲
Retrofit	0.9714 ± 0.0010 ▲	**0.9893 ± 0.0000** ★	0.9844 ± 0.0010 ▲	0.9847 ± 0.0000 ▲	0.9775 ± 0.0010 ▲	**0.9885 ± 0.0000** ★	0.9833 ± 0.0010 ▲	0.9838 ± 0.0000 ▲
ZXing	0.9880 ± 0.0010 ▲	0.9857 ± 0.0000 ▲	0.9920 ± 0.0010 ▲	**0.9946 ± 0.0000** ★	**0.9954 ± 0.0010** ★	0.9869 ± 0.0000 ▲	0.9919 ± 0.0010 ▲	0.9946 ± 0.0000 ▲
TIME BUDGET: 80%								
DSpace	0.9505 ± 0.0010 ▲	**0.9825 ± 0.0070** ★	0.9717 ± 0.0010 ▲	0.9501 ± 0.0000 ▲	0.9636 ± 0.0010 ▲	**0.9810 ± 0.0080** ★	0.9714 ± 0.0010 ▲	0.9494 ± 0.0000 ▲
Druid	0.6683 ± 0.1100 ▲	**0.9820 ± 0.0040** ★	0.9431 ± 0.0110 ▲	0.9584 ± 0.0020 ▲	0.6957 ± 0.1080 ▲	**0.9902 ± 0.0050** ★	0.9447 ± 0.0110 ▲	0.9512 ± 0.0010 ▲
Fastjson	0.8925 ± 0.0110 ▲	0.9296 ± 0.0340 ▲	**0.9687 ± 0.0010** ★	0.9659 ± 0.0000 ▲	0.9155 ± 0.0110 ▲	0.9320 ± 0.0320 ▲	0.9690 ± 0.0010 △	**0.9693 ± 0.0000** ★
Deeplearning4j	0.6620 ± 0.0190 ▲	0.8641 ± 0.0010 ▲	0.8896 ± 0.0020 ▲	**0.8964 ± 0.0010** ★	0.8563 ± 0.0120 ▲	0.7991 ± 0.0010 ▲	0.8576 ± 0.0040 ▲	**0.8613 ± 0.0010** ★
Guava	0.5316 ± 0.0050 ▲	**0.5679 ± 0.0010** ★	0.5224 ± 0.0030 ▲	0.5218 ± 0.0000 ▲	0.5334 ± 0.0040 ▲	**0.5700 ± 0.0010** ★	0.5225 ± 0.0030 ▲	0.5217 ± 0.0000 ▲
IOF/ROL	0.3043 ± 0.1170 ▲	0.7036 ± 0.0030 ▲	0.6898 ± 0.0030 ▲	**0.7142 ± 0.0000** ★	0.3181 ± 0.0920 ▲	0.6795 ± 0.0030 ▲	0.6712 ± 0.0030 ▲	**0.6939 ± 0.0000** ★
Lexisnexis	0.9571 ± 0.0080 ▲	0.9842 ± 0.0150 ▼	0.9862 ± 0.0010 △	**0.9868 ± 0.0000** ★	0.9683 ± 0.0110 ▲	0.9826 ± 0.0150 ▼	0.9864 ± 0.0010 △	**0.9871 ± 0.0000** ★
Paint Control	0.9160 ± 0.0000 ▲	**0.9172 ± 0.0000** ★	0.9074 ± 0.0000 ▲	0.9074 ± 0.0000 ▲	0.9157 ± 0.0000 ▲	**0.9177 ± 0.0000** ★	0.9074 ± 0.0000 ▲	0.9074 ± 0.0000 ▲
OkHttp	0.8564 ± 0.0050 ▲	**0.9478 ± 0.0000** ★	0.8407 ± 0.0020 ▲	0.8545 ± 0.0000 ▲	0.8979 ± 0.0030 ▲	**0.9362 ± 0.0000** ★	0.8397 ± 0.0020 ▲	0.8509 ± 0.0000 ▲
Retrofit	0.9747 ± 0.0030 ▲	**0.9916 ± 0.0000** ★	0.9873 ± 0.0010 ▲	0.9873 ± 0.0000 ▲	0.9809 ± 0.0030 ▲	**0.9903 ± 0.0000** ★	0.9858 ± 0.0010 ▲	0.9862 ± 0.0000 ▲
ZXing	0.9879 ± 0.0000 ▲	**0.9996 ± 0.0000** ★	0.9927 ± 0.0010 ▲	0.9953 ± 0.0000 ▲	0.9945 ± 0.0020 ▲	**0.9996 ± 0.0000** ★	0.9926 ± 0.0010 ▲	0.9952 ± 0.0000 ▲

A. *Practical Implications*

Use of LSTM: we observed in the last section that the LSTM network is more suitable for low budget environments. For less restrictive budgets, COLEMAN and RF outperform LSTM with the exception of 3 systems: Fastjson, DSpace and LexisNexis; for those the LSTM presented consistent results and high scores in all time budgets. Regarding Fastjson and DSpace, both systems have low failure distribution, and the test case volatility does not have influence or correlation with the failing test cases. Considering the LexisNexis dataset, the failing test cases are distributed in many tests, and this corroborates to a better performance in higher time budgets (less restriction).

On the other hand, LSTM and the other algorithms had low performance in the dataset IOF/ROL. Figure 5 illustrates, for this system, the test case volatility and failures by cycle along with the CI Cycles[4]. As we can observe, the test case volatility is high, as well as the number of failing tests. The high test case volatility may be associated with a pre-commit strategy, for instance, test case selection. Consequently, the algorithms do not have enough historical information to make accurate predictions; the low performance of the LSTM network may be related with a *catastrophic forgetting* situation.

Another consideration is that a low number of test cases per cycle may be the cause for a low score in the time budget of 10%, for example, in the dataset OkHttp. In cases where there is low volatility and a low number of test cases, like in the ZXing dataset, a high score was achieved probably maybe due to the historical data relevance as the CI cycles continue.

Use of RF: the analysis in the training phase shows the sliding window size W has a significant impact on the overall performance of the model. Another observation is that the prioritization time is directly correlated with the number of estimators given. This time doubles for every 250 estimators.

RF obtained high values of NAPFD and APFDc for all time budgets in the systems: Fastjson, Druid, LexisNexis and Deeplearning4j. For Deeplearning4j, RF probably achieved a good score due to the low amount of test cases. But RF did not obtain high NTR values. In most cases the difference for the best NTR values obtained by the other algorithms is small, like for the Druid dataset. But for Okhttp dataset it is significant. In conclusion, RF is better suited for low and mid budget (10% and 50%).

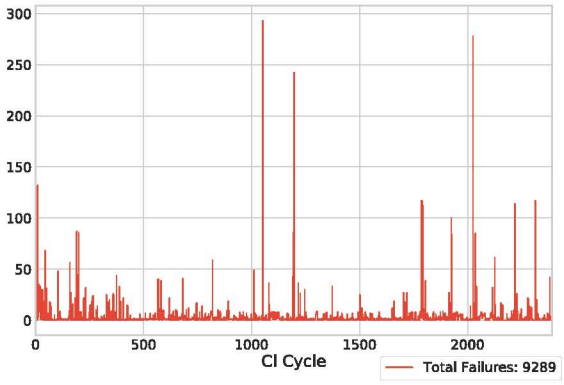
RF has the best performance for more restrictive budgets, and low prioritization time. This assumption could differ in case of a larger dataset with more rich historical test data, in which the neural network would have enough data to learn with more accuracy.

Use of COLEMAN and RETECS: COLEMAN presented the best performance for less restrictive budgets. This probably happens due to a better adaptation in these cases. RETECS performed well only for APFDc for low and mid budgets.

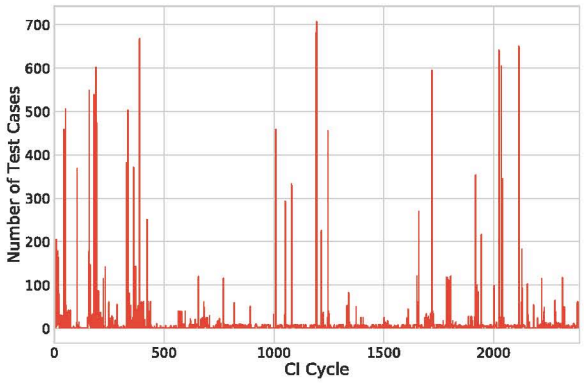
B. *Threats to Validity*

Below we present possible threats to the validity of our results, according to the taxonomy of Wohlin et al. [43].

[4]See the datasets' characteristics in our supplementary material [14]



(a) Failures By Cycle.



(b) Test Case Volatility.

Fig. 5: IOF/ROL System Overview.

Internal Validity: the execution environments and parameter settings can be considered a threat. As the configuration of the algorithm was selected by experimentation, not all possibilities were covered, as such, a better configuration could lead to better results. The usage of the CuDNNLSTM framework restricts the customization of the hidden layer activation functions. In cases where its use is not required and hardware resources are abundant, more personalized configurations of activation functions can yield more promising results.

External Validity: we used only eleven systems. Thus, the results cannot be generalized. Experiments with other systems should be performed. To this end, we believe that our study can be replicated, using the raw data present in our repository [14] and using the configuration description for LSTM and RF.

Conclusion Validity: the randomness of the ML algorithms is a threat. To minimize this threat, we selected for our analysis the results of 10 executions. Another threat is related to the statistical tests used. To minimize this threat, we used tests commonly adopted for non-deterministic algorithms in software engineering problems. Lastly, the multiple environments for executions could also induce some randomness due to different hardware configurations.

VII. Concluding Remarks

This work introduced a TCPCI approach based on the failure history of the test cases. The approach uses the sliding window method for the application of ML regression techniques, which allows dealing properly with some particularities of the CI environments: EvE dilemma and volatility of test cases. The approach is lightweight, requires only the past tests' results; it is model-free and independent of the programming language; and does not require code analysis.

The approach was evaluated using the regression algorithm RF and an LSTM network with eleven systems and three time budgets. The results show that both, RF and LSTM, apply to the TCPCI problem regarding the build time and the CI cycles.We performed a comparison with the state-of-the-art RL approaches COLEMAN and RETECS, regarding some common regression testing measures. Regarding NAPFD, all systems and budgets, RF reached the best values, or statistically equivalent to the best ones, in 23 cases (out of 33, ≈70%), while COLEMAN and LSTM presented the best values respectively, in 20 (≈61%) and 18 (≈55%) cases, and RETECS only 8 (≈24%). For APFDc, RF and COLEMAN reached the best values, or statistically equivalent, in 19 cases (out of 33, ≈58%), while RETECS 18 (≈55%) and LSTM 16 (≈48%).

We discuss some practical implications for the usage of the algorithms and approaches evaluated. RF outperforms the other algorithms for low and mid budgets (budgets of 10% and 50%). COLEMAN outperforms the other algorithms for the less restrictive budget (of 80%). The LSTM network has good performance for the budget of 10%, and RETECS when the cost function (APFDc) is considered, for low and mid budgets.

Future work includes the evaluation of the approach with other systems, as well as exploring the use of other features regarding the test cases to obtain a more accurate prediction. Other ML algorithms should be further evaluated.

Acknowledgment

This research was partially funded by CAPES (grants: 88887.501291/2020-00, 88882.382199/2019-01, 88887.501314/2020-00) and CNPq (grant: 305968/2018-1).

References

[1] B. Fitzgerald and K.-J. Stol, "Continuous software engineering: A roadmap and agenda," *Journal of Systems and Software*, vol. 123, pp. 176–189, January 2017.

[2] J. A. Prado Lima and S. R. Vergilio, "Test case prioritization in continuous integration environments: A systematic mapping study," *Information and Software Technology*, vol. 121, pp. 106–268, 2020.

[3] S. Yoo and M. Harman, "Regression Testing Minimization, Selection and Prioritization: A Survey," *Software Testing, Verification & Reliability*, vol. 22, no. 2, pp. 67–120, Mar. 2012.

[4] S. Elbaum, G. Rothermel, and J. Penix, "Techniques for Improving Regression Testing in Continuous Integration Development Environments," in *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2014. New York, NY, USA: ACM, 2014, pp. 235–245.

[5] C. Henard, M. Papadakis, M. Harman, Y. Jia, and Y. Le Traon, "Comparing White-box and Black-box Test Prioritization," in *Proceedings of the 38th International Conference on Software Engineering*, ser. ICSE'16. New York, NY, USA: ACM, 2016, pp. 523–534.

[6] A. Bertolino, A. Guerriero, B. Miranda, R. Pietrantuono, and S. Russo, "Learning-to-rank vs ranking-to-learn: Strategies for regression testing in continuous integration," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ser. ICSE'20. New York, NY, USA: Association for Computing Machinery, 2020, p. 1–12.

[7] H. Spieker, A. Gotlieb, D. Marijan, and M. Mossige, "Reinforcement Learning for Automatic Test Case Prioritization and Selection in Continuous Integration," in *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2017. New York, NY, USA: ACM, 2017, pp. 12–22.

[8] J. A. Prado Lima and S. R. Vergilio, "A multi-armed bandit approach for test case prioritization in continuous integration environments," *IEEE Transactions on Software Engineering*, p. 12, 2020.

[9] T. G. Dietterich, "Machine learning for sequential data: A review," in *Structural, Syntactic, and Statistical Pattern Recognition*, T. Caelli, A. Amin, R. P. W. Duin, D. de Ridder, and M. Kamel, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 15–30.

[10] H. Robbins, "Some aspects of the sequential design of experiments," *Bulletin of the American Mathematical Society*, vol. 58, no. 5, pp. 527 – 535, 1952.

[11] L. Breiman, "Random forests," *Machine Learning*, vol. 45, no. 1, pp. 5–32, 2001.

[12] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Comput.*, vol. 9, no. 8, p. 1735–1780, Nov. 1997. [Online]. Available: https://doi.org/10.1162/neco.1997.9.8.1735

[13] X. Qu, M. B. Cohen, and K. M. Woolf, "Combinatorial Interaction Regression Testing: A Study of Test Case Generation and Prioritization," in *IEEE International Conference on Software Maintenance*, oct 2007, pp. 255–264.

[14] E. A. Roza, J. A. Prado Lima, R. C. Silva, and S. R. Vergilio, "Supplementary Material - Machine Learning Regression Techniques for Test Case Prioritization in Continuous Integration Environment," January 2022, URL https://osf.io/bek98/?view_only=4e9ea3b608924998aeb3f5fb80baa5f2.

[15] D. Marijan, A. Gotlieb, and S. Sen, "Test Case Prioritization for Continuous Regression Testing: An Industrial Case Study," in *IEEE International Conference on Software Maintenance*. IEEE, Sep. 2013, pp. 540–543.

[16] D. Marijan, "Multi-perspective Regression Test Prioritization for Time-Constrained Environments," in *Proceedings of the 2015 IEEE International Conference on Software Quality, Reliability and Security*, ser. QRS'15. Washington, DC, USA: IEEE Computer Society, 2015, pp. 157–162.

[17] D. Marijan, M. Liaaen, A. Gotlieb, S. Sen, and C. Ieva, "TITAN: Test Suite Optimization for Highly Configurable Software," in *Proceedings of the IEEE International Conference on Software Testing, Verification and Validation*, ser. ICST. IEEE, Mar. 2017, pp. 524–531.

[18] D. Marijan, A. Gotlieb, and M. Liaaen, "A learning algorithm for optimizing continuous integration development and testing practice," *Software: Practice and Experience*, vol. 49, no. 2, pp. 192–213, 2019.

[19] L. Xiao, H. Miao, and Y. Zhong, "Test case prioritization and selection technique in continuous integration development environments: a case study," *International Journal of Engineering & Technology*, vol. 7, no. 2.28, pp. 332–336, 2018.

[20] A. Haghighatkhah, M. Mäntylä, M. Oivo, and P. Kuvaja, "Test prioritization in continuous integration environments," *Journal of Systems and Software*, vol. 146, pp. 80–98, 2018.

[21] B. Busjaeger and T. Xie, "Learning for Test Prioritization: An Industrial Case Study," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2016. New York, NY, USA: ACM, 2016, pp. 975–980.

[22] X. Li, H. Jiang, Z. Ren, G. Li, and J. Zhang, "Deep learning in software engineering," *ArXiv: 805.04825*, 2018.

[23] C. Watson, N. Cooper, D. N. Palacio, K. Moran, and D. Poshyvanyk, "A systematic literature review on the use of deep learning in software engineering research," *ArXiv: 805.04825*, 2020.

[24] Y. Yang, X. Xia, D. Lo, and J. Grundy, "A survey on deep learning for software engineering," *ArXiv: 2011.14597*, 2020.

[25] N. Medhat, S. M. Moussa, N. L. Badr, and M. F. Tolba, "A framework for continuous regression and integration testing in IoT systems based on deep learning and search-based techniques," *IEEE Access*, vol. 8, pp. 215 716–215 726, 2020.

[26] M. Raeisi Nejad Dobuneh, D. Jawawi, M. Vahidi-Asl, and M. Malakooti, "Clustering test cases in web application regression testing using self-organizing maps," *International Journal of Advances in Soft Computing and its Applications*, vol. 7, pp. 1–14, 01 2015.

[27] L. Xiao, H. Miao, T. Shi, and Y. Hong, "LSTM-based deep learning for spatial–temporal software testing," *Distributed and Parallel Databases*, vol. 38, p. 687–712, May 2020.

[28] S. Pouyanfar, S. Sadiq, Y. Yan, H. Tian, Y. Tao, M. P. Reyes, M.-L. Shyu, S.-C. Chen, and S. S. Iyengar, "A survey on deep learning: Algorithms, techniques, and applications," *ACM Computer Survey*, vol. 51, no. 5, Sep. 2018. [Online]. Available: https://doi.org/10.1145/3234150

[29] Z. Yu, F. Fahid, T. Menzies, G. Rothermel, K. Patrick, and S. Cherian, "TERMINATOR: better automated UI test case prioritization," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. FSE. ACM, 2019, pp. 883–894.

[30] J. A. Prado Lima and S. R. Vergilio, "Supplementary material - a multi-armed bandit approach for test case prioritization in continuous integration environments," 2019. [Online]. Available: https://osf.io/epnuk/

[31] G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold, "Test Case Prioritization: An Empirical Study," in *Proceedings of the IEEE International Conference on Software Maintenance*, ser. ICSM '99. IEEE Computer Society, 1999, pp. 179–188.

[32] W. H. Kruskal and W. A. Wallis, "Use of Ranks in One-Criterion Variance Analysis," *Journal of the American Statistical Association*, vol. 47, no. 260, pp. 583–621, 1952.

[33] H. B. Mann and D. R. Whitney, "On a test of whether one of two random variables is stochastically larger than the other," *The annals of mathematical statistics*, pp. 50–60, 1947.

[34] M. Friedman, "A comparison of alternative tests of significance for the problem of m rankings," *The Annals of Mathematical Statistics*, vol. 11, no. 1, pp. 86–92, 1940.

[35] A. Vargha and H. D. Delaney, "A Critique and Improvement of the CL Common Language Effect Size Statistics of McGraw and Wong," *Journal of Educational and Behavioral Statistics*, vol. 25, no. 2, pp. 101–132, jan 2000.

[36] S. Chetlur, C. Woolley, P. Vandermersch, J. Cohen, J. Tran, B. Catanzaro, and E. Shelhamer, "cuDNN: Efficient primitives for deep learning," 2014, arXiv:1410.0759.

[37] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015, software available from tensorflow.org. [Online]. Available: https://www.tensorflow.org/

[38] F. Chollet *et al.*, "Keras," https://keras.io, 2015.

[39] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[40] M. Berman, J. S. Chase, L. Landweber, A. Nakao, M. Ott, D. Raychaudhuri, R. Ricci, and I. Seskar, "Geni: A federated testbed for innovative network experiments," *Computer Networks*, vol. 61, pp. 5 – 23, 2014, special issue on Future Internet Testbeds Part I. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1389128613004507

[41] L. Guo, Y. Ma, B. Cukic, and H. Singh, "Robust prediction of fault-proneness by random forests," in *15th International Symposium on Software Reliability Engineering*, 2004, pp. 417–428.

[42] S. Hartshorn, "Machine learning with random forests and decision trees," *Kindle edition*, 2016.

[43] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén, *Experimentation in Software Engineering: An Introduction*. Kluwer Academic Publishers, 2000.