2023 38th IEEE/ACM International Conference on Automated Software Engineering Workshops (ASEW) | 979-8-3503-3032-8/23/$31.00 ©2023 IEEE | DOI: 10.1109/ASEW60602.2023.00014

Neural Network-based Test Case Prioritization in Continuous Integration

Andreea Vescan
Babes-Bolyai University,
Department of Computer Science
M. Kogalniceanu 1,
Cluj-Napoca, Romania
andreea.vescan@ubbcluj.ro

Radu Găceanu
Babes-Bolyai University,
Department of Computer Science
M. Kogalniceanu 1,
Cluj-Napoca, Romania
radu.gaceanu@ubbcluj.ro

Arnold Szederjesi-Dragomir
Babes-Bolyai University,
Department of Computer Science
M. Kogalniceanu 1,
Cluj-Napoca, Romania
arnold.szederjesi@ubbcluj.ro

***Abstract*—In continuous integration environments, the execution of test cases is performed for every newly added feature or when a bug fix occurs. Therefore, regression testing is performed considering various testing strategies. The Test Case Prioritization (TCP) approach considers reordering test cases so that faults are found earlier with a minimum execution cost.**

The purpose of the paper is to investigate the impact of neural network-based classification models to assist in the prioritization of test cases. Three different models are employed with various features (duration, fault rate, cycles count, total runs count) and considering information at every 30 cycles or at every 100 cycles.

The results obtained emphasize that the NEUTRON approach finds a better prioritization with respect to NAPFD (normalized average percent of the detected fault) than random permutation and is comparable with the solutions that used either duration or faults, considering that it combines both values. Compared to other existing approaches, NEUTRON obtains similar competitive results when considering a budget of 50% and the best results when considering budgets of 75% and 100%.

***Index Terms*—Test Case Prioritization, Continuous Integration, Neural Network, Faults, Duration, Cycles**

I. Introduction

A crucial step in software development is testing, which helps to ensure the usability, reliability, and robustness of a system by identifying bugs or discrepancies from the intended functionality. Regression testing is one particular kind of testing that is carried out to make sure that previously developed and tested software continues to function as intended after a change is made. Modifications, additions, or deletions to the operating system, database, or code are examples of changes. Regression testing seeks to find any bugs that might have unintentionally been introduced in the codebase during these changes. It is critical to ensure the long-term effectiveness of software since it helps to ensure that earlier program functionalities continue to work after incorporating the new changes.

Regression testing, while extremely beneficial, also presents a number of challenges. Due to frequent updates and modifications to the codebase, the regression test suite can become large and resource-intensive over time. Additionally, it may not be possible or feasible to run all test cases within the time constraints of a typical development cycle [1]. Test Case Prioritization (TCP) provides a solution to these issues by ordering the test cases in a manner that increases the probability of early defect detection. In TCP, the execution of test cases is prioritized based on several criteria such as: code coverage, frequency of code changes, the criticality of the software module, and historical failure rate [2]. By applying this process, it is ensured that even with limited time and resources, the most significant potential issues are evaluated first, thus enabling faster detection and correction of defects. This can considerably improve the entire software development process, which may also impact the overall functionality and quality of the software system.

Despite its practical importance, regression testing remains a challenging task due to its inherent trade-off between effectiveness and efficiency, as well as difficulties in accurately capturing the dependencies in complex software systems. Thus, the regression testing problem is worth exploring, being an omnipresent one during the software development life cycle. It is essential to find efficient and effective test case executions with the aim of obtaining higher qualitative systems.

Moreover, testing within Continuous Integration (CI) environments comes with a unique set of challenges [3]. First, unlike typical testing methods, CI demands tight control over both the selection and the prioritization of the most promising test cases, that is, those that are most likely to detect early failures. Second, it could be argued that selecting test cases that execute the most recent code changes could be a good approach. However, when dealing with system tests, the traceability links between code and test cases may not always be available or easily accessible. Third, due to time constraints in each cycle, not all tests can be executed, necessitating an effective method for test case selection. These difficulties suggest the need for alternative approaches in CI testing.

Several state-of-the-art approaches from the literature like [4], [5] and [6] propose online methods (e.g. based on reinforcement learning) for the test case prioritization problem in CI contexts, thus eliminating the training phase. Nevertheless, since significant volumes of data are still needed by these methods in order to become effective, not needing a training

This work was funded by the Ministry of Research, Innovation, and Digitization, CNCS/CCCDI - UEFISCDI, project number PN-III-P1-1.1-TE2021-0892 within PNCDI III.

phase is, in our opinion, not necessarily an advantage. In our approach, we use a neural network to prioritize test cases in continuous integration environments. Experiments are carried out on industrial datasets, some of which are provided in [4]. We transform the original datasets and we significantly reduce their size *without any loss of data*. For example, one of the datasets is reduced from 32261 lines to 1962 lines. Additionally, we enrich the datasets with additional features. All datasets are publicly available, so they can be used by other researchers. We built three classifiers trained on historical data about test case execution: the first classifier uses the same features from the original datasets, and the second and third ones use additional features that aggregate information about the test cases from several groups of CI cycles. We compare our results with standard deterministic methods as well as with state-of-the-art approaches from the literature, obtaining better results in some of the considered use cases.

The contributions of this paper are as follows:

- significantly reducing the size of the datasets from [4], *without any loss of information*
- augmenting the datasets with more features describing finer-grained details regarding the test case executions
- labeling the datasets by using the joined capabilities of both AI and human domain experts
- a neural network-based approach for testing in CI contexts, that takes into account historical execution data regarding test cases, as well as their duration
- proposing three classification models: the first one used the original features and all test cases of training, the other two classifiers are trained on data containing aggregated information from several groups of CI cycles
- evaluating the proposed models on industrial datasets.

The paper is structured as follows. Section II presents background information on the Test Case Prioritization problem and its time-limited variant that occurs in the continuous integration context. In Section III, the related work on test case prioritization is presented, including state-of-the-art approaches that address the continuous integration environment. Section IV, presents our approach including the classification models, the proposed transformations of the dataset (reducing the size without any information loss, extending the feature set, labeling process), and the metrics used for analysis and evaluation. Section V presents the design of the experiments, and the results are presented in Section VI. The potential benefits of our approach and threats to validity are presented in Section VII and Section VIII, respectively. Section IX, summarizes the contributions of this paper, presents the results, and draws concluding remarks.

II. TCP Problem

According to [7], the Test Case Prioritization (TCP) problem is formally defined as follows:

Definition 1: **Test Case Prioritization [7]:** a test suite, T, the set of permutations of T, PT; a function from PT to real numbers, f. The goal is to find T ∈ PT such that:

$$(\forall T'')(T'' \in PT)(T'' \neq T')[f(T') \geq f(T'')] \qquad (1)$$

From Definition 1, the function f assigns a real value to a permutation of T according to the test adequacy of the particular permutation.

Unfortunately, this problem formulation does not capture the notion of a time limit to execute the test suite. An extended problem formulation, Time-limited Test Case Prioritization, augments the conventional Test Case Prioritization problem by introducing a time constraint. This added element suggests that due to time restrictions, there might be situations where not all test cases can be run. It is noteworthy that not only time but also other factors could impose limitations on the test case selection process. However, the formulation given below can be adapted without any loss of generality.

Definition 2: **Time-limited Test Case Prioritization Problem (TTCP) [4]**: find T ∈ PT such that:

$$f(T) \geq f(T')\wedge$$
$$\sum_{t_k \in T'} t_k.duration \leq M \wedge$$
$$\sum_{t_k \in T} t_k.duration \leq M, \forall T' \in PT,$$

where:

- $t_k.duration$ represents the duration of a test case k from a test suite
- M is the maximum time available for the test suite execution.

In this paper, since links between code changes and test cases are assumed not to be available, historical information about test case execution needs to be considered. This is why we use the following definition for Adaptive Test Case Selection Problem (ATCS) in our approach.

Definition 3: **Adaptive Test Case Selection Problem (ATCS) [4]**: let $T_1, T_2, \ldots T_{k-1}$ be a sequence of previously executed test suits, find T_k such that $f(T_i)$ is maximized and $\sum_{t_k \in T} t_k.duration \leq M$.

III. Related work

Test case selection and prioritization is an intensively investigated topic in the literature, and it has been approached in several ways by addressing different optimization goals and by using a plethora of techniques in doing so [8]–[13]. According to the machine learning technique involved, test case selection and prioritization studies could be classified into the following categories: supervised learning, unsupervised learning, reinforcement learning, and natural language processing [14].

Recent studies apply reinforcement learning (RL) techniques to Test Case Prioritization (TCP) in Continuous Integration (CI) due to RL's ability to adapt to CI's dynamic nature without needing full retraining. Once trained, the RL agent can evaluate a test case, assign it a score, and use that score to order or prioritize the test cases. Although most of the RL studies consider only the execution history to train their agent, we have found one study, namely [9], where, in addition to the execution history, code complexity metrics have also been used. In their very comprehensive research paper, the authors of [9] evaluated and compared 10 machine learning algorithms, focusing on the comparison between supervised learning and reinforcement learning for the Test Case Prioritization (TCP) problem. Experiments are performed on six publicly available datasets, and on the basis of the results, the authors propose several guidelines for applying machine learning to regression testing in Continuous Integration (CI). In [9], the authors also propose some new metrics (Rank Percentile Average (RPA) and Normalized-Rank-Percentile-Average (NRPA)) for evaluating how close a prediction ranking is to the optimal one. Unfortunately, in [14] it is explained in detail that the NRPA metric is not always suitable. Nevertheless, the paper by [9] remains a very thorough and elaborate research study, and the authors of [14] include it in a short list of research papers that are actually reproducible.

The main idea of clustering in the TCP context is the assumption that test cases with similar characteristics, like coverage and other attributes, are likely to have comparable fault detection abilities. Many papers, such as, for example, [10], utilized the K-means algorithm or variations of the K-means algorithm. The Euclidean distance is the most commonly used similarity measure in clustering, but there are some attempts to use other similarity measures like the Hamming distance. The paper from [11] is an example in this sense. In this study, the authors represent the coverage information for a test case as binary strings, where each bit indicates whether or not a source code element has been covered by a test. An interesting approach is proposed in [12] where the authors use clustering for anomaly detection of passing and failing executions. The key idea is that failures tend to be grouped into small clusters, while passing tests will group into larger ones, and their experiments suggest that their hypothesis is valid.

Supervised learning is probably one of the most commonly used ML techniques to address TCP as a ranking problem. Specifically, these techniques typically use one of three distinct ranking models for information retrieval: pointwise, pairwise, and listwise ranking. In [9], the authors used a state-of-the-art ranking library [15] and evaluated the effectiveness of Random Forest (RF), Multiple Additive Regression Tree (MART), L-MART, RankBoost, RankNet, Coordinate ASCENT (CA) for TP. Their results show that (MART) is the most accurate model. Although supervised learning can achieve high accuracy, a major issue is that a full dataset should be available before training. In order to support incremental learning, the model often needs to be rebuilt from scratch, which is time-intensive, and hence not quite ideal for CI.

The use of NLP for TCP seems to be quite limited. The core motivation to apply NLP techniques is to exploit information in either textual software development artifacts (e.g., bug description) or source code that is treated as textual data. In general, the idea is to transform test cases into vectors and then to compute the distance between pairs of test cases. The test cases are then prioritized using different strategies. An interesting approach is proposed in [13], where NLP is used to pre-process the specifications that describe the components of the system under test. Then they used recurrent neural networks to classify the specifications into the following components: user device, protocols, gateways, sensors, actuators, and data processing. On the basis of this classification, test cases belonging to these standard components were selected. Then, they used search-based approaches (genetic algorithms and simulated annealing) to prioritize the selected test cases.

In the following, some studies are presented that specifically target the CI context. In the approach from [16], the authors use the sliding time window to choose the test suits to be applied in a pre-submit phase of testing by tracking their history. In a subsequent post-submit phase, a similar approach is employed to prioritize tests. Experiments with the Google Shared Dataset of Test Suite Results (GSDTSR) indicate that the testing load is reduced and delays in fault detection are reduced.

The paper [4] introduces an innovative method for prioritizing and selecting test cases in Continuous Integration (CI) environments. The proposed method employs reinforcement learning to select and order test cases based on their duration, last previous execution, and history of failure. In the study, both a tableau-based agent and a network-based agent are employed as reinforcement learning agents. The tableau-based agent operates using a tabular structure to associate the different states of the CI system with their respective actions, maintaining an action-value (Q-table) that updates based on the outcomes of previous test cases. On the contrary, the network-based agent uses artificial neural networks (ANN) to accomplish the same task. In this setup, the ANN serves as the function approximator for the agent's policy, i.e., its strategy for selecting actions based on the current state. In contrast to the tableau-based agent, which stores the expected rewards for each (state, action) pair in a lookup table, the network-based agent generalizes across similar states using the ANN. This strategy enables the agent to manage larger state spaces and continuous actions, making it more scalable and adaptable. Several industrial datasets were used to evaluate the efficacy of the study. The paper also provides a comparative analysis against deterministic test case prioritization methods. The results show that the proposed approach can learn to prioritize test cases in 60 cycles starting with a model-free memory and no past information regarding test cases. This result is comparable to fundamental deterministic test case prioritization techniques, which indicates that it is a promising method for CI test case prioritization.

In [5] a Multi-Armed Bandit (MAB) approach for test case prioritization in CI environments is presented. The MAB problems are a class of sequential decision problems that may be seen as a simplified form of RL. As opposed to RL, they do not need context information and the actions do not change environment states. Also, MAB does neither need to handle the state space nor to use function approximators. The authors performed extensive experiments and evaluated their approach with several parameter configurations, and with some parameter settings, they were able to outperform the approach from [4].

In [6], an approach based on the Dueling Bandit Gradient Descent (DBGD) algorithm is introduced. Evaluation of several industrial datasets indicates that after 150 cycles, the proposed method outperforms other state-of-the-art approaches.

Most approaches from related work in CI use online models for prioritization and advocate that this choice is an advantage for the continuous integration context because the training phase is eliminated. Nevertheless, in order to obtain comparable results with deterministic methods, rather significant volumes of data are still needed (e.g. at least 60 cycles, that is, about two months of data, in case of one of the state-of-the-art methods [4]). The approach of [6], which is to our knowledge the best in the literature, needs 150 cycles (so about 5 months of data) in order to achieve this result. Considering the vast volumes of data (and time span) needed by these methods to become efficient, we argue that eliminating the training phase is not necessarily such a great advantage as it may seem. Our approach, which uses a neural network, needs, of course, a training phase, but our experiments show that it outperforms related approaches in some of the studied use cases. In our experiments, we investigate several scenarios, some of which involve transformations of the original datasets enriching them with additional features, in order to improve the performance of our model.

IV. NEUTRON APPROACH FOR TCP IN CI

In this section, we outline our NEUTRON (NEUral network based Test case pRioritization in cOntinuous iNtegration) approach for test case prioritization in continuous integration environments by employing test case priority based on neural network classification that incorporates knowledge about previous executions of the test cases and time execution.

A. Approaches in Test Case Priority in Continuous Integration

Regression testing may be implemented using various strategies regarding test suite execution, from *retest all* to test suite minimization (TSM), test case selection (TCS), and test case prioritization (TCP).

Figure 1 graphically depicts the three approaches that we are going to investigate in this paper: (1) the *retest all* approach using random execution of test cases in the test suite, (2) the *sort by fails* approach that used the information regarding test case failures in all cycles to order the test suite, and (3) the NEUTRON approach (NEUral network based Test case pRioritization in cOntinuous iNtegration) that uses the priority-based classification for each test case. As in the definition in Section II, given $T = (t_1, t_2, ..., t_n)$, *retest all* with random strategy means generating a random permutation of T, while *sort by fails* refers to sorting the test suite based on the number of failures in all cycles. The neural network classification-based TCP in CI means using the test cases priority classification to order them from the most critical to the least critical and then selecting the test cases that are included in the specific time execution budget.

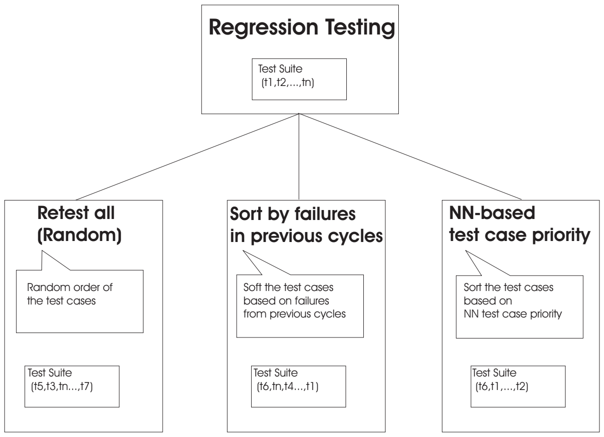
Regression Testing
Test Suite (t1,t2,...,tn)
Retest all (Random)
Random order of the test cases
Test Suite (t5,t3,tn...,t7)
Sort by failures in previous cycles
Sort the test cases based on failures from previous cycles
Test Suite (t6,tn,t4...,t1)
NN-based test case priority
Sort the test cases based on NN test case priority
Test Suite (t6,t1,...,t2)

Fig. 1. TCP in Continous Integration approaches

In what follows, we provide details on our NEUTRON approach for TCP in CI using the NN-based test case priority classification.

B. Test case priority classification based on Neural Network

Our overall approach to TCP using NN-based classification consists of three phases: data preparation, training, and testing. Figure 2 depicts the process.

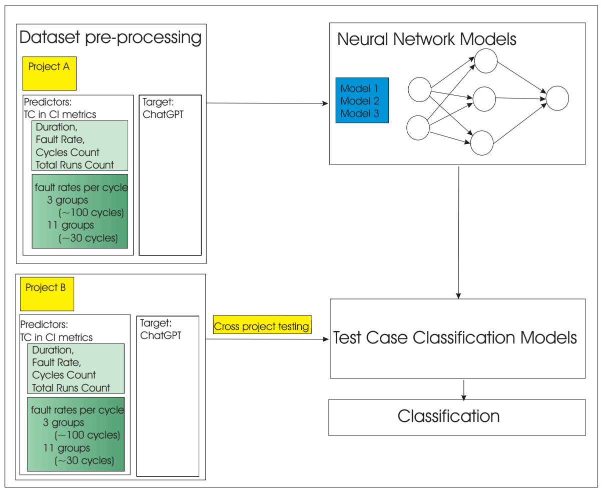
Dataset pre-processing
Project A
Predictors: TC in CI metrics
Duration, Fault Rate, Cycles Count Total Runs Count
fault rates per cycle 3 groups (~100 cycles) 11 groups (~30 cycles)
Target: ChatGPT
Project B
Predictors: TC in CI metrics
Duration, Fault Rate, Cycles Count Total Runs Count
fault rates per cycle 3 groups (~100 cycles) 11 groups (~30 cycles)
Target: ChatGPT
Cross project testing
Neural Network Models
Model 1 Model 2 Model 3
Test Case Classification Models
Classification

Fig. 2. Overview of the neural networks-based models for TCP in Continous Integration

The data preparation steps comprise the computation for each test case of the number of failures in each cycle, the number of failures in a cycle over the number of total runs in that cycle, along with the duration of that test case, and the $FaultRate$, namely the average of all failures in all cycles. More information on this data preparation step is provided in Section IV-C.

In the training step, various NN models are employed in order to classify the priority of the test cases considering information from previous executions like the number of failures, the total number of executions, and the execution time. The models used different projects with different characteristics to be trained. Details regarding this aspect are provided in Section IV-E and Section IV-C.

The testing phase consists of applying the obtained models to other projects (cross-project) as shown graphically in Figure 2. For example, we train the model using data from *Project A* and test the model on data from *Project B*.

In what follows, we outline and describe in detail the constituent elements of the NN approach, starting with the used and constructed dataset, followed by feature information, employed models, and finishing with the metrics used for analysis.

C. Dataset

Three industrial datasets, namely two from ABB Robotics Norway [1] (Paint Control and IOF/ROL, for testing complex industrial robots) and Google Shared Dataset of Test Suite Results (GSDTSR) [2] are used in this study. All three datasets contain information about historical test case executions, along with the verdicts (pass, fail), with CI cycles over 300. The two ABB datasets are split into daily intervals, whereas GSDTSR is split into hourly intervals as it originally provides log data of 16 days. However, the average test suite size per CI cycle in GSDTSR exceeds that in the ABB datasets. An overview of the dataset is presented in Table I.

TABLE I
DATASETS INFORMATION OVERVIEW

Project name	**Test cases information**			
	No. of Test Cases	***CI cycles***	***Verdicts***	***Failed***
Paint Control	89	352	25,594	19.36%
IOF/ROL	1941	320	30,319	28.43%
GSDTSR	5,555	336	1,260,617	0.25%

Additionally, in order *to structure the information around a test case and not around cycle or execution*, we extracted and reorganized the information provided in the three datasets, thus characterizing each test case with execution cycles, fails versus passes, etc.

Three matrices (the cycle matrix, the fault matrix, and the verdict matrix) are obtained following the reorganization of the initial dataset, for each matrix we provide next the constituent elements:

- cycle matrix, containing:
  - Testcase - id of the test case
  - Duration - execution time for the given testcase
  - Fault Rate - represents the number of total faults over the number of total runs
  - Cycles count - the number of cycles in which the test case was executed
  - Total Runs Count - sum of the runs of the given test case over all cycles
  - Cycle 1..N - number of faults over the number of runs in the given cycle for the given test case
- fault matrix (as we did not have a list of faults, we considered that each test case would detect one unique fault if the test case ever failed), containing:
  - test case ids as rows
  - fault ids as column
  - value cells containing 1 if the given test case detected the given fault, 0 otherwise
- verdict matrix, containing:
  - Testcase - id of the test case
  - Cycle $\{1...N\}$ - last verdict for the given test case and cycle, "success" means pass, "fail" means that the test case did not pass and found a fault.

The constructed datasets with the built matrices are available at this link [17].

D. Features

The features used in our investigations are, for each test case:

- duration
- number of runs in all cycles
- number of total executions
- fault rate
- a rate between the number of fails of the test case in a cycle over the number of runs in that cycle.

The constructed datasets with the above-specified features are based on the datasets in the investigation in the paper [4].

The sequence of test cases within the TCP paradigm is typically decided by a domain expert. However, in the current scenario, due to the extensive data volume, we opted to utilize the services of ChatGPT to establish the order. ChatGPT is already *listed as an author* in some research papers like [18], [19]. While many researchers disapprove [20] with this, using it as an assistant for certain tasks is strongly promoted in several areas of activity including biomedical research and healthcare [21], software testing education [22], software architecture and design [23], source code generation [24]. We leveraged the AI's proficiency and potential insights regarding TCP to procure a prioritized array of test case ids. In regards to how we prompted the LLM (Large Language Model), after providing the dataset, and carefully explaining what is the format of the input and what we expect as an output, we used the following prompt: "As a testing, especially regression testing, expert I want you to prioritize all testcases starting with the most important one." Furthermore, in order to facilitate computational operations on the enumerated test case identifiers, each ID was associated with a real value within the range of 0 to 1. The most optimal test case received a score of 1.0. The score for each successful test case was determined by deducting quantity 1 divided by the total count of test cases from the score of the previous test case.

Two methods were used to verify the ChatGPT labelling following an exhaustive analysis of the dataset's constitution and the features' semantics, namely, one manual and one using

[1] https://new.abb.com/products/robotics
[2] https://bitbucket.org/HelgeS/atcs-data/src

the NAPFD metric. First, the authors randomly selected and inspected 30 test cases and established, based on their expertise, that the test cases appeared to be correctly prioritized. It is important to mention that, since we trained the neural network using supervised learning on the IOF/ROL dataset (because it was the most balanced dataset according to the test case verdicts), the ChatGPT labels were only needed for this. Subsequently, we computed the NAPFD metric (see Section IV-F). For all the methods considered (see Section V), given a budget of 100%, we achieved values close to 1, indicating a highly accurate ordering.

E. Models

The models employed in our study used a subset of the existing features based on the constructed datasets as mentioned in Section IV-C. It is worth mentioning that all the models employed used all instances of the IOF/ROL project as training and the other two projects, namely, *Paint Control* and *GSDTSR* are used for the testing phase.

The similarities and differences between the models considering the used features are provided next:

- *Model 1* uses four features from the dataset: Duration, Fault Rate, Cycles Count, and Total Runs Count.
- *Model 2*: uses four features from the dataset (Duration, Fault Rate, Cycles Count, Total Runs Count) augmented by other three features regarding the average of faults at every approximately 100 cycles (namely, first 107 cycles, next 107 cycles and the last 106 cycles).
- *Model 3*: uses four features from the dataset (Duration, Fault Rate, Cycles Count, Total Runs Count) augmented by other 11 features regarding the average of faults every approximately 30 cycles (namely, 10 groups of 30 cycles and the last group of 20 cycles).

As shown in Figure 2, the three models use various features as predictors, the 4 features, in the first rectangle in the Predictors section, are used in *Model 1*, and for the other 2 models, *Model 1* and *Model 2* the fault rates from various groups of cycles are used (3 groups for the second model and eleven groups for the third model). For the testing phase of *Model 2* the following cycle groups were used for the *paintcontrol* project 118, 118, 116, and for the *gsdtsr* project 112, 112, 112. For the testing phase of *Model 3*, for th *paintcontrol* project 32 cycles for used for each group and for the *gsdtsr* project 31 cycles of each group, except the last one with 26 cycles.

F. Metrics for Analysis

The APFD metric [25] is defined as follows (n is the number of test cases, m is the number of faults, and TF_f represents the position of the test case in the prioritized test suite that detects the fault f). The higher the value of the APFD score, the better the fault detection.

$$APFD = 1 - \frac{TF_1 + TF_2 + ...TF_m}{m \times m} + \frac{1}{2 \times n} \quad (2)$$

Normalized APFD (NAPFD) [26] is an extension of APFD in order to incorporate the fact that not all test cases are executed and failures can be undetected. When p=1, namely, all faults are detected, the NAPFD is the same as the APFD formulation.

NAPFD = p - TF$_1 + TF_2 + ...TF_m \frac{}{m \times m + \frac{p}{2 \times n}}$(3)

where p = the number of faults detected by the prioritized test suite divided by the number of faults detected in the full test suite.

V. Design of Experiments

Research investigation consists of various experiments. Figure 3 graphically shows the overview of the experiments performed. All experiments considered a minimum of four features, namely, *Duration*, *Fault Rate*, *Cycles count*, and *Total runs count*. The first and second experiments also considered the information regarding faults for each test case for each cycle. In the first experiment, we follow the experimental protocol considered in [4] (where data from every 30 CI cycles were used, considering 50% budget) and [6] (where 50%, 75%, 95% budgets were considered), thus considering in our experiment information provided every approximately 30 cycles. In the second experiment, we considered information provided every approximately 100 cycles. And in the last experiment, we only considered the four general information about the test cases. For all three experiments, we performed various budget percentages, from 25% to 50% and 75%, and also 100%.

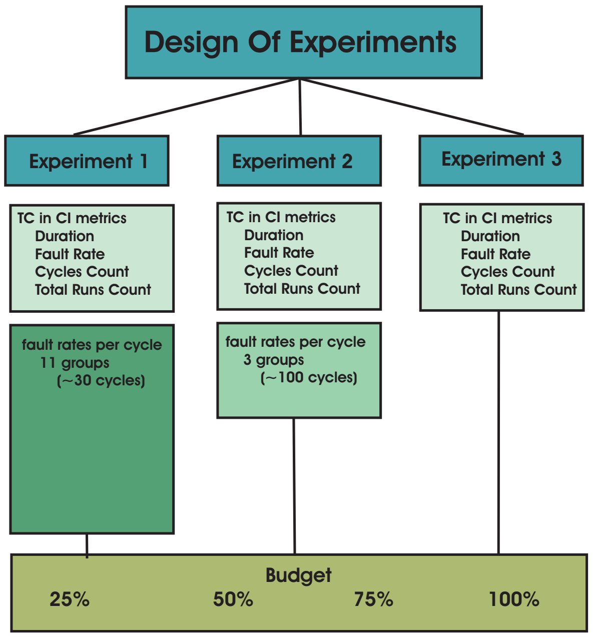

Fig. 3. Design of experiments

VI. Results

This section outlines the results of the above-mentioned experiments.

For all experiments, the training was carried out in the *IOF/ROL* project (since it was the most balanced dataset among the available ones) and testing was carried out in the *Paint Control* and *GSDTSR* projects.

A. Experiment 1

The current experiment considers for each test case the four general features mentioned above, together with the fault information for each cycle. The neural network uses the provided data for every 30 cycles.

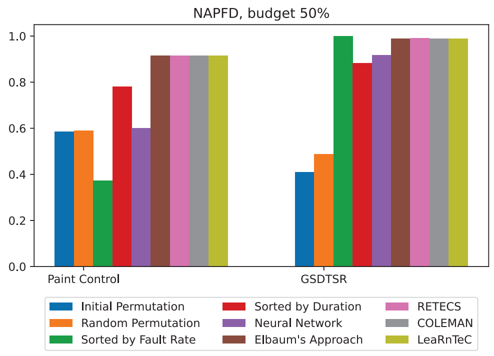

Fig. 4. Experiment 1 with approximately 11x30 cycles, considering 50% budget

As can be observed in Figure 4, the best results, in the case of 50% budget, are obtained by RETECTS [4] and COLEMAN [5]. It should be noted that NEUTRON obtains better results than the *Random* solutions and also better than *Sorted by Duration* and *Sorted by Fault Rate* in each of the tested projects, however, the NEUTRON solution embedded both features regarding duration and fault rate.

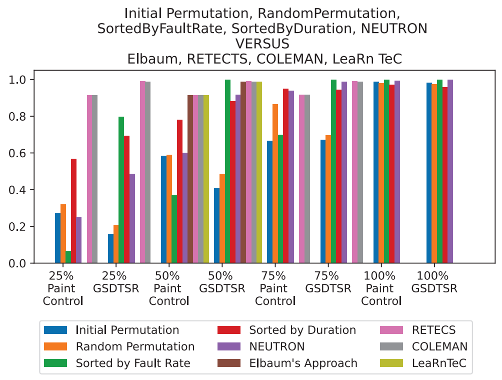

Fig. 5. Experiment 1 with approximately 11x30 cycles, considering all budgets

In Figure 5, it is shown that NEUTRON obtains the best solution for 75% budget in the case of testing *Paint Control* and for both testing projects in the case of the 100% budget. Table II contains the values of the NAPFD obtained for the previous papers Elbaum's approach [16], RETECTS [4], COLEMAN [5], LeanRnTeC [6] and the results for *NEUTRON* approach, along with the results for the initial permutation, random and soft by duration or faults. It should be stated that the complete values exist only for the 50% budget. For the 25% and 75%, we have considered in the table the results of previous solutions for 10% and 80% respectively.

B. Experiment 2

The second experiment considers, like the first experiment, for each test case the four general features along with fault information regarding each cycle, at every 100 cycles.

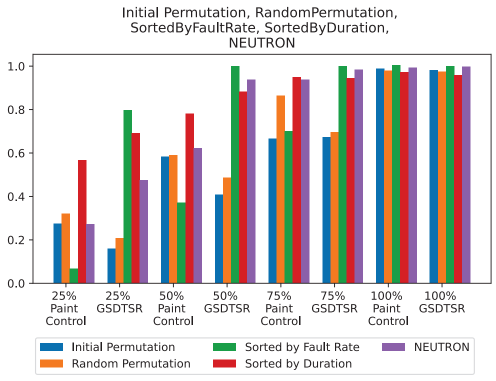

Fig. 6. Experiment 2 with approximately 3x100 cycles, considering all budgets

In Figure 6, it can be seen that the best solutions are obtained for the *Sorted by Fault* or *Sorted by Duration*. However, it must be stated that the *NEUTRON* solution combines both features, namely, faults and duration, obtaining comparable results when only considering one of the dimensions.

C. Experiment 3

This third experiment takes into account only the four general features.

In Figure 7, the same results are found, namely *NEUTRON* finds better solutions than the *Random* permutation, however, considering only single feature, the *Sorted by Faults Rate* and *Sorted by Duration* obtain better ones. Furthermore, in this case, it should be noted that the *NEUTRON* approach simultaneously considers both features (duration and faults) and the results are comparable in this case.

D. Comparisons of the three models

We have also investigated the three models proposed in the context of the 50% budget. As shown in Figure 8, in the case of the *Paint Control* project the best model is for the one trained with no cycles followed by the Model with 100 cycles. In the case of the *GSDTSR* project, the best model is with 100 cycles followed by the one with 30 cycles.

TABLE II
EXPERIMENT 1

Project	NAPFD								
	Initial	Random	Sort Fault Rate	Sort Duration	NEUTRON	Elbaum	RETECS	COLEMAN	LeaRnTeC
25%-Paint Control	0.275280	0.320224	0.067415	0.567415	0.251276		0.915	0.915	
25%-GSDTSR	0.160602	0.208414	0.798651	0.692944	0.486858		0.9911	0.9893	
50%-Paint Control	0.584485	0.589887	0.372999	0.780898	0.600487	0.9145	0.915	0.915	0.915
50%-GSDTSR	0.409727	0.486987	0.999755	0.882094	0.917503	0.9891	0.9911	0.9893	0.9894
75%-Paint Control	0.666328	0.864736	0.700434	0.949570	0.937796		0.9162	0.9171	
75%-GSDTSR	0.672442	0.697674	0.999878	0.945154	0.987035		0.9921	0.9893	
100%-Paint Control	0.988700	0.980494	1	0.972667	0.994003				
100%-GSDTSR	0.982327	0.974485	0.999904	0.959203	0.998308				

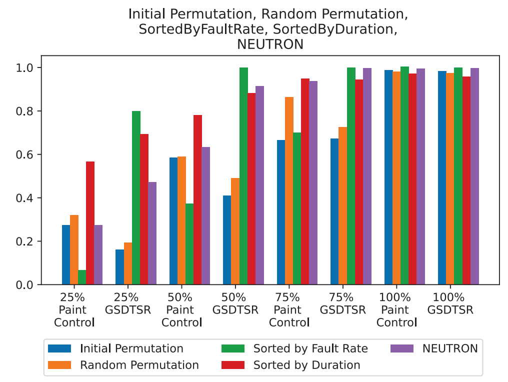

Fig. 7. Experiment 3 with four features, considering all budgets

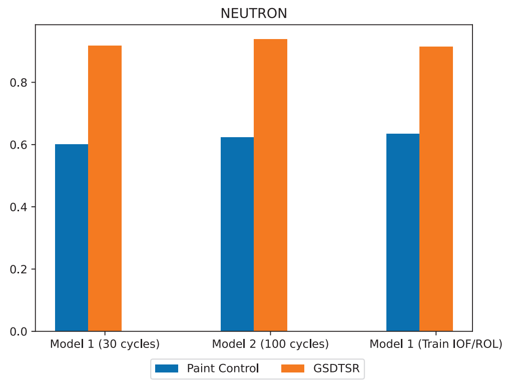

Fig. 8. Comparisons of the three models

VII. Potential Benefits of the Contribution

In regression testing, there is no time to run all existing test cases, so there are various strategies to execute the tests with the highest impact, finding more faults as early as possible in the time execution. Test Case Prioritization overcomes some of the drawbacks of the selection or reduction mechanism by not discarding test cases. In this regard, one benefit of this contribution is that the results of the models are provided for various time budgets, thus allowing the software tester to select the proper model according to the needs and time available.

Another benefit of the approach is that it allows the identification of faults early during the testing process since it could be applied not only in the early stages of development/testing when only a few tests are implemented, but also in the later stages of development when the test suite is very large.

Regarding the constructed dataset, various researchers could use it to further propose other methods and compare it with other existing ones. Thus, in this respect, the benefit will be to the researchers being able to further analyze and compare new solutions with state-of-the-art ones. Furthermore, considering that the datasets were industrial, this newly constructed dataset augmented with more features describing finer-grained details regarding the test case executions could be used to validate other similar industrial projects.

In future work, the best model could be embedded in an IDE tool such that developers and testers could benefit from the prioritization of test cases when specifying a time budget. It would also be possible to specify how many cycles to be considered when aiming to find a fault or when aiming at a specific pass/fail execution history.

VIII. Threats to Validity

Experiments may be vulnerable to certain threats to validity, and the outcome of the research is influenced by various aspects. Next, several points are indicated that may have influenced the results obtained, stating the action taken to mitigate them.

Internal. The primary internal validity threat is the use on ChatGPT to label the datasets. However, we have explained our validation approach for this labeling process and we have confidence that it is accurate. Moreover, both the datasets and the labels are public, so anyone at any point can inspect and validate them. Another potential threat to validity is the presence of bugs in our implementation. However, through comprehensive testing and code review, we hope to have significantly reduced the likelihood of such issues.

Construct. The original datasets contain few features with respect to the test cases. To mitigate this threat, we enhanced the datasets with additional features, but we believe that more and more diverse information can be included in the datasets.

External. We evaluate our approach on three industrial datasets. Clearly, we should have used more datasets, but to our knowledge, there are no other datasets that have the required data, especially historical information regarding the execution of the test cases. Also, another important aspect related to the structure of the dataset and the existing information is the scenario where a single test case may discover a single fault and thus a fault is found by a single test case. Even if, in theory, a test case was designed with the purpose to check a behavior, it is possible that the same fault is identified by various test cases. Thus, more complex datasets are needed in the context of continuous integration environments, also with links between the changed and associated test case or/and requirements that are checked.

IX. Conclusions

In continuous integration settings, a set of tests is run for each new feature or bug fix. There are several strategies to select or prioritize the execution of the tests. Test Case Prioritization reorders the test cases so that faults are found earlier with a minimum execution cost.

The paper investigated the use of neural network-based classification models to help prioritize tests. Three different models were employed with various features (duration, fault rate, cycles count, total runs count) and considering information at every 30 cycles or at every 100 cycles.

The NEUTRON approach finds a better prioritization with respect to NAPFD than random permutation. The NEUTRON results are comparable with other sort-based solutions that used either duration or faults; in addition, it considers both features when constructing the solution. Compared to other existing state-of-the-art approaches, NEUTRON achieves similar competitive results when considering a budget of 50% and the best results when considering budgets of 75% and 100%.

In future work, further experiments are going to be performed with more projects and with more complex scenarios considering various cycles and more test cases discovering the same faults, along with the connection to the change in the source code or in the requirements.

Acknowledgment

This work was funded by the Ministry of Research, Innovation, and Digitization, CNCS/CCCDI - UEFISCDI, project number PN-III-P1-1.1-TE2021-0892 within PNCDI III.

References

[1] C. Paul and D. Jorgensen, Software Testing: A Craftsman'sApproach. Auerbach Publishers, Incorporated, 2021.

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

[3] J. A. Prado Lima and S. R. Vergilio, "Test case prioritization in continuous integration environments: A systematic mapping study," Inf. Softw. Technol., vol. 121, no. C, may 2020. [Online]. Available: https://doi.org/10.1016/j.infsof.2020.106268

[4] 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: Association for Computing Machinery, 2017, p. 12–22. [Online]. Available: https://doi.org/10.1145/3092703.3092709

[5] J. A. P. Lima and S. R. Vergilio, "A multi-armed bandit approach for test case prioritization in continuous integration environments," IEEE Transactions on Software Engineering, vol. 48, no. 2, pp. 453–465, 2022.

[6] S. Omri and C. Sinz, "Learning to rank for test case prioritization," in 2022 IEEE/ACM 15th International Workshop on Search-Based Software Testing (SBST), 2022, pp. 16–24.

[7] T. L. Graves, M. J. Harrold, J. Kim, A. Porters, and G. Rothermel, "An empirical study of regression test selection techniques," in Proceedings of the 20th International Conference on Software Engineering, 1998, pp. 188–197.

[8] R. Pan, T. A. Ghaleb, and L. Briand, "Atm: Black-box test case minimization based on test code similarity and evolutionary search," arXiv preprint arXiv:2210.16269, 2022.

[9] 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. [Online]. Available: https://doi.org/10.1145/3377811.3380369

[10] Z. Khalid and U. Qamar, "Weight and cluster based test case prioritization technique," 2019 IEEE 10th Annual Information Technology, Electronics and Mobile Communication Conference (IEMCON), pp. 1013–1022, 2019.

[11] P. Kandil, S. Moussa, and N. Badr, "Cluster-based test cases prioritization and selection technique for agile regression testing," Journal of Software: Evolution and Process, vol. 29, no. 6, p. e1794, 2017, e1794 JSME-15-0111.R1. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/smr.1794

[12] R. Almaghairbe and M. Roper, "Separating passing and failing test executions by clustering anomalies," Software Quality Journal, vol. 25, no. 3, p. 803–840, sep 2017. [Online]. Available: https://doi.org/10.1007/s11219-016-9339-1

[13] 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.

[14] G. T. e. a. Pan R., Bagherzadeh M., "Test case selection and prioritization using machine learning: a systematic literature review," Empir Software Eng, vol. 29, pp. 1 – 43, 2022.

[15] V. Dang and M. Zarozinski, "Ranklib," 2020. [Online]. Available: https://sourceforge.net/p/lemur/wiki/RankLib/

[16] 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: Association for Computing Machinery, 2014, p. 235–245. [Online]. Available: https://doi.org/10.1145/2635868.2635910

[17] A. Vescan, R. Gaceanu, and A. Szederjesi-Dragomir, "Neural network-based test case prioritization in continuous integration (neutron) - dataset," figshare, 2023. [Online]. Available: https://doi.org/10.6084/m9.figshare.23727300.v1

[18] S. O'Connor and ChatGPT, "Open artificial intelligence platforms in nursing education: Tools for academic progress or abuse?" Nurse Education in Practice, vol. 66, p. 103537, 2023. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1471595322002517

[19] T. H. Kung, M. Cheatham, ChatGPT, A. Medenilla, C. Sillos, L. D. Leon, C. Elepaño, M. Madriaga, R. Aggabao, G. Diaz-Candido, J. Maningo, and V. Tseng, "Performance of chatgpt on usmle: Potential for ai-assisted medical education using large language models," medRxiv, 2022. [Online]. Available: https://www.medrxiv.org/content/early/2022/12/21/2022.12.19.22283643

[20] C. Stokel-Walker, "ChatGPT listed as author on research papers: many scientists disapprove," Nature, vol. 613, no. 7945, pp. 620–621, January 2023. [Online]. Available: https://ideas.repec.org/a/nat/nature/v613y2023i7945d10.1038_d41586-023-00107-z.html

[21] D.-Q. Wang, L.-Y. Feng, J.-G. Ye, J.-G. Zou, and Y.-F. Zheng, "Accelerating the integration of chatgpt and other large-scale ai models into biomedical research and healthcare," MedComm –

Future Medicine, vol. 2, no. 2, p. e43, 2023. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/mef2.43

[22] S. Jalil, S. Rafi, T. D. LaToza, K. Moran, and W. Lam, "Chatgpt and software testing education: Promises perils," in 2023 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW), 2023, pp. 4130–4137.

[23] A. Ahmad, M. Waseem, P. Liang, M. Fahmideh, M. S. Aktar, and T. Mikkonen, "Towards human-bot collaborative software architecting with chatgpt." New York, NY, USA: Association for Computing Machinery, 2023. [Online]. Available: https://doi.org/10.1145/3593434.3593468

[24] L. Jacques, "Teaching cs-101 at the dawn of chatgpt," vol. 14, no. 2, 2023. [Online]. Available: https://doi.org/10.1145/3595634

[25] G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold, "Test case prioritization: An empirical study," in Proceedings IEEE International Conference on Software Maintenance-1999 (ICSM'99).'Software Maintenance for Business Change'(Cat. No. 99CB36360). IEEE, 1999, pp. 179–188.

[26] X. Qu, M. Cohen, and K. Woolf, "Combinatorial interaction regression testing: A study of test case generation and prioritization," in 2007 IEEE International Conference on Software Maintenance. Los Alamitos, CA, USA: IEEE Computer Society, oct 2007. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/ICSM.2007.4362638