A Method for Regression Testing Plan Ordering for Non-Automated Executions in Black Box Testing

Vinicius Hernandes[1], André Carvalho[1], Eulanda Santos[1], Yan Soares[1], Hygo Oliveira[1], Adamor Barros[1], Ronaldo Soares[1], Alexandre Lima[1] Raoni Ferreira[2], Gabriel Martins[2], Lucas Carvalho[2], Nicolas Assumpção[3], José Nascimento[3], Eliane Collins[4], Silvia Ascate[4], Mateus Souza[4]

[1] Universidade Federal do Amazonas (UFAM)

[2]Universidade Federal do Acre (UFAC)

[3]Motorola Mobility LLC

[4]Instituto de Desenvolvimento Tecnológico

{vmh, andre, emsantos, yan.soares}@icomp.ufam.edu.br

{hygo.sousa, adamorb, ronaldo.soares, alexandre.lima}@icomp.ufam.edu.br

{raoni.ferreira, lima.gabriel, lucas.hildelbrano}@sou.ufac.br

{nicolas, josern}@motorola.com

{eliane.collins, silvia.ascate, mateus.souza}@indt.org.br

Abstract. *In this paper, we propose a method for prioritizing regression test cases based on the probability of detecting software execution failures without source code analysis. To achieve this, our method employs the SentenceBERT model to extract embeddings from textual information of development commits and test scripts. These embeddings are then used by machine learning models to predict the probability of detecting a failure. Our experiments show that the proposed method achieves results equal to or better than those of human experts in 92.52% to 94.24% of scenarios when evaluating the APFD (Average Percentage Faults Detected) metric, an overall gain of 10% in APFD mean and a potential gain of up to 6.03% in test plan prioritization counting cases.*

1. Introduction

During the software development process, various testing routines are executed to ensure the quality of the product being developed. Among these routines, regression testing aims to verify whether changes made to components introduce new defects into the product by re-executing existing test cases to verify that previously working functionalities remain unaffected. Regression testing can be divided into the following stages: analysis of modified components, selection of suitable test cases, ordering of test case execution, and generation of documentation for the test results. [Rizwan et al. 2022].

In the first stage, the testing team analyzes information about product changes and scales plans to cover functional modifications executed in the new version of a product, as well as dependencies between these components [Zhang et al. 2022]. After identifying potentially affected components by the product change comes the selection of test cases

to be executed. As test cases typically possess some form of tagging and cataloging, information about existing test case scopes is crossed with scope information for affected components during the selection stage. Consequently, test cases are selected based on defined criteria, such as covering affected components and generating a set of test cases capable of identifying the highest number of errors in the product [Pan et al. 2020].

This process affects the effectiveness of the testing plan in finding existing product failures. Usually, this procedure is executed by a group of testers where different levels of expertise among team members complement each other, often ensuring good performance.

Once the test case set has been defined, the subsequent stage involves determining their order of execution. In this stage, despite the numerous possible approaches to define their execution sequence, a common objective exists: to execute tests that are likely to uncover existing failures first. Typically, this ordering is performed based on the experience of the tester or historical information about the execution of test cases [Pan et al. 2020], which in turn affects the efficiency of the plan, given its direct impact on the speed at which failures will be discovered. Finally, after determining the ordering strategy, the tests are executed, and their results are recorded. These results may lead to the implementation of further changes in the software, which will subsequently enter the cycle and repeat until the product is delivered with all necessary corrections to ensure proper functioning.

In this context, test cases serve as units of execution for testing plans and can be categorized based on two characteristics: their level of automation [Tahvili et al. 2018] and functional or structural observation [Mafra et al. 2009]. Considering automation, tests are divided into automated and non-automated tests. Automated tests are executed by a computer in a controlled environment, without human intervention. In contrast, non-automated tests rely on a human tester for their full or partial execution [Tahvili et al. 2018].

In terms of functional characteristics, test cases are categorized as black-box or structural (white-box) based on distinct product features that they observe. While white-box testing is entirely dependent on the implementation of source code and examines the correctness of implementation aspects such as logical decisions, data structures, and loop structures, black-box testing verifies the product's functional capabilities without directly observing the source code [Mafra et al. 2009]. This allows black-box testing to be produced before the functionality is implemented, provided that only the test specification is based on the proposed product features.

In the specific context of automation characteristics, there is a significant interest in automating test case execution due to various reasons. These reasons include attempts to exclude human error during execution and the possibility of increasing the test scale with reduced costs [Rizwan et al. 2022]. However, the automation process can have a very high cost, both in terms of resources [Mafra et al. 2009] and time frames, or may even be impossible to implement. Due to this scenario, many companies still perform manual testing, even though, despite following an execution routine, the tester's interpretation may lead to divergent results [Mafra et al. 2009].

Regression testing cycles are justified by ensuring product quality with respect

to execution correctness, which is an indispensable step in various product development scenarios. Due to this necessity, reducing its execution cost, whether financially or temporally, confirms the interest in optimizing resource allocation for the test task [Al-Sabbagh et al. 2021][Mehta et al. 2021][Ramírez et al. 2023]. However, human analysis can be inefficient when seeking the best possible resource utilization, either due to selecting redundant tests (unnecessarily increasing testing effort) or a biased execution order (resulting in potential defect correction delays).

In this perspective, machine learning techniques can be adopted to ensure confidence in the quality of the proposed test order for the execution of regression tests [Palma et al. 2018], [Wu et al. 2019]. This approach reduces the cost of generating testing plans and allows for the rapid analysis of massive data sources [Mehta et al. 2021]. According to [Rizwan et al. 2022], the use of machine learning techniques (ML) has been suitable and successful in automating many testing steps, such as product change identification, test case selection, and ordering. Moreover, advances in natural language processing (NLP) research have shown that these techniques have achieved improved performance in extracting information as well as in categorizing natural language texts, which may be difficult for machines to interpret due to their ambiguity and lack of structure [Tahvili et al. 2018].

Considering this context, this work proposes a framework for ordering regression testing plans, observing non-automated and black-box test cases. The proposed ordering is based on the probability of classification of a test case execution as a failure in a specific product. Therefore, the higher its probability of failure, the higher the priority of executing a test. An important feature of the method is that we use textual information without requiring access to the source code of the tested product. This enables suggesting an ordering of a testing plan according to the probabilities of failure of a specific test case.

The framework obtains the probability of a test execution resulting in a failure by deploying ML techniques, as follows: a classification model is trained to observe the information from a test case and a product change record, associating this tuple with a failure or not. The association is learned from the historical registry of executions, whose data allow us to associate a past execution of a test case with a specific product and its corresponding result. Finally, due to the textual nature of test cases and available product change records, it is necessary to use ML and NLP techniques to obtain codified information useful for the classification model.

The remainder of this work is structured as follows: Section 2 describes some related work relevant to our research problem. Section 3 details the proposed framework. In Section 4, we present the results achieved, along with an explanation of the data employed. Finally, in Section 5, a general summary of the results achieved is presented, as well as future works to be carried out.

2. Related Work

The ordering of test cases is a crucial step in the software testing process, allowing testers to focus on the most important tests based on predefined criteria. Consequently, testing time and resources are utilized more efficiently, increasing the probability of detecting critical failures in the software as quickly as possible. Various techniques for prioritizing test cases have been proposed in the literature, each with its own strengths and weaknes-

ses. This section presents a review on this subject, focusing on approaches most relevant to the work developed here.

In [Mafra et al. 2009], the author proposed a tool that selects a set of ordered test cases according to their relevance. The selection is based on metrics obtained from previous execution results and metrics obtained from analyzing the content of each test case against the product scope. These metrics determine a relevance score for each test case, making it possible to identify the most relevant tests within a testing plan.

The authors in [Zhang et al. 2022] highlight that regression testing selection (RTS) optimizes testing by executing only the subset of tests that can be affected by changes. In this context, they showed that RTS based on ML works well in large code repositories. They developed a combination of RTS based on analysis and RTS based on ML to learn the impact of code changes on test results. Experiments were conducted on a training dataset obtained through mutation analysis. Compared to RTS based on analysis, this study showed that the combination with RTS based on ML selects about 25% fewer test cases, improving the task of prioritizing test cases to be evaluated.

In addition to selecting sets of test cases (TCs), another approach used in the literature is defining the order of execution for TCs to improve test execution time. In this sense, [Palma et al. 2018] developed an evaluative study of TCs from a dataset of open-source projects developed in Java, which compose the defects4j dataset. The authors proposed a logistic regression model for predicting TC priority, with results submitted to traditional quality metrics such as: Traditional Historical Fault Detection Metric (TM), Method Coverage (MC), Changed Method Coverage (CMC), and Size of Tests (ST); and quality metrics based on similarity, such as: Basic Counting (BC), Edit Distance (ED), Hamming Distance (HD), and Improved Basic Counting (IBC). The results indicated that the predictive model suggested by the authors can perform test case prioritization.

The work presented in [Spieker et al. 2017] proposes the RETECS method, focusing on automatically selecting and prioritizing TCs in the context of continuous software integration. The authors emphasize the relevance and difficulty of selecting TCs with higher probability of generating errors due to uncertainty generated by commit information regarding the impact of code changes. RETECS selects and prioritizes TCs using reinforcement learning and neural networks. To evaluate the performance of this method, the authors implemented a normalized APFD metric and identified an agent-based neural network as the best predictor for prioritizing TCs.

In [An and Yoo 2022], the authors also focused on selecting and prioritizing TCs. The Fault Localization (FL) technique was used, which measures the impact of each TC on the test plan suite (TP). In this work, it is reinforced that it is essential that FL performance be directly related to TP-tested test quality. To expand the TCs that comprise each TP, the authors developed the Fault Diagnosability Gain metric, which explores continuous FL results to emphasize parts of the program requiring more information.

Reinforcement learning techniques were also employed in [Omri and Sinz 2022]. In this work, the challenge of prioritizing TCs was approached as an online ranking problem to mitigate the scalability impact in continuous integration environments. The authors validated the proposed method in a case study industrial setting, showing that more than 95% of test failures were reported to software engineers, while only 40% of

available TCs were executed.

Finally, in [Shankar and Sridhar 2024] it is highlighted that information loss is the biggest difficulty in prioritizing TCs in continuous integration environments. This loss significantly impacts the regression task performed by tools already produced in the literature. To resolve this problem, the authors developed DeepRP, a Q-learning-based reinforcement learning model that better understands features extracted from source code, version control, and code coverage.

It is worth mentioning that most of the work described in this section refers to the prioritization or selection of TCs related to continuous integration cycles through software versioning. In each cycle, metadata about the tests is generated through logs, test case execution time, instructions that guide the test cases, as well as coding of the version control system. In almost all observed methods, the use of the developed source code is used in the approaches proposed by the other solutions. On the other hand, in this work, the ordering of TCs is defined based on metadata contained in test scripts, commits and historical data related to the tested product, without observing the product's source code itself. In this way, our tool seeks to understand what was modified in the product from the textual artifacts mentioned above. This type of ordering, called black box, demonstrated compatibility and reliability through the APFD metric.

3. Proposed Method

In this work, we propose a framework that aims to optimize the order of execution of test cases to efficiently identify existing failures in product functionality, thereby ensuring a wider time margin for correcting these failures. To achieve this objective, it is necessary to prioritize the execution of test cases within a testing plan related to the changes made in the product. Therefore, there is an implicit interest in executing test cases that will reject the product as early as possible. Consequently, we seek to define an execution order that contains failure situations at the top positions.

To select the execution order, ML models are used to identify the probability of the product failing a test: a characteristic we call *failure probability*. Once the failure probability is calculated, the ordering of test cases is entirely based on this new characteristic. To calculate the failure probability, we provide the ML model with a feature vector composed of information about both the test case and the tested product version. This is done by concatenating two separately extracted feature vectors, one containing features extracted from the test case and another containing features representing the product. To characterize the product, we used commit texts from changes made to that version of the product, while for test case characteristics, we used the test script text. We did so since we believe that NLP ML models can extract characteristics from both to identify possible product failures, since the commits contain information regarding product changes and the test scripts of the tested characteristics.

It is important to note that both product and test case data are natural language texts, making it very challenging to extract computable information efficiently to characterize this textual information. To obtain a feature vector that can represent both sets of textual information, we employ neural networks based on transformer architecture SBERT [Reimers 2019], where the network provides a feature vector expressing the information existing in a text sentence. Therefore, given a product and a testing plan, we

are able to establish a sequence of steps (illustrated in Figure 1) capable of ordering test cases based on failure probability. These steps are detailed in the next subsections.

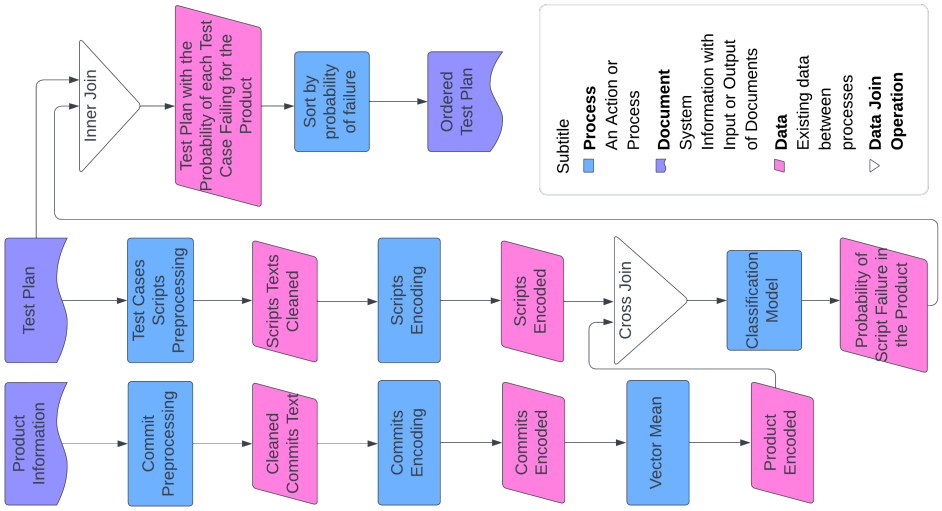

Figura 1. Flowchart of the Ordering Framework

3.1. Product Information and Test Case Preprocessing

The product and testing plan information are abstractions for the entire set of relevant information related to the problem used in our framework. As mentioned earlier, the product information is extracted from commit texts generated during the development of the referred product version. On the other hand, the information in the test plan is the test cases that make it up, each test case has a script of action instructions (test script) to be applied to the product. Both types of information possess textual characteristics close to natural language texts. Therefore, the first step for the proper use of this information involves text preprocessing.

Given the similarity in text type between both data sources, the preprocessing step in both cases consists of filtering out very common or very rare terms and removing known irrelevant texts. For example, during this stage, attention warnings, multiple blank spaces, special characters such as brackets, numbers, HTML tags, among others were removed. In our work, the filters used were essentially based on regular expression searches (REGEX). However, due to the specificities of each textual input (commits and test scripts), it was necessary to develop specific filters for each type of input. Once preprocessing is completed, the next step involves text encoding, in which texts are converted into numerical representation vectors.

3.2. Product Information and Test Case Encoding

The encoding models are responsible for observing textual content and transforming its representation into a numerical vector. The information contained in the vectors generated by the encoding models enables differentiation between texts based on the original textual content, representing it in a numerical form. The encoding model employed in our work was SBERT (Sentence Bidirectional Encoder Representations for Transformers).

3.3. SBERT

The Bidirectional Encoder Representations for Transformers (BERT) encoder is based on a transformer network, where the textual input is encoded into a 768-dimensional vector representation when the standard version of the architecture is used. This representation takes into account not only the content of existing terms in the text but also the arrangement of these terms within the text, as illustrated in Figure 2. SBERT is an extension of BERT, designed specifically for generating sentence representations. The key distinction between them is that SBERT trains BERT on a specific task of sentence encoding—rather than predicting words in a sentence. As a result, SBERT produces full-sentence representations, making them useful for tasks such as text-based information retrieval and text clustering [Reimers 2019].

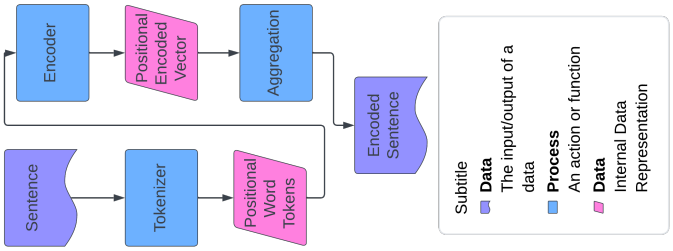

Figura 2. Flowchart of the SBERT Process.

In this work, we employed SBERT to encode the two textual entities: test case scripts and commit messages. The preprocessed texts were then submitted to the pretrained SBERT model, which transformed the test steps and commits into embeddings. Our goal is that these embeddings condense the semantic content of the texts in a compact yet informative manner. Additionally, the vectorial representation of textual data enables various ML methods to perform tasks with the features extracted by the SBERT network. The next stage of our framework focuses on using some ML methods for the task of test case classification.

3.4. Classification of Test Cases Applied to a Product

As previously mentioned, the ordering of test cases is performed based on the probability of failure for each test case. This probability is calculated at this stage as follows: The encoding step of test case scripts and products generates a 1536-position vector (768 to represent the product and 768 to represent the test case) for each instance of test cases. These vectors are used as input for classification algorithms. Subsequently, the classification algorithm analyzes the received vector and returns the probability that this vector represents an accepted or failed test case, using the class probability output of the model.

Various ML techniques can be employed at this stage, since the objective of the algorithm is to learn to correlate the input vectors with the failure probability of the test case. In this work, the investigated methods were as follows: KNN (K-nearest neighbors) [Batista et al. 2009], Logistic Regressor [Brzezinski and Knafl 1999], MLP (Multi-Layer Perceptron) [Murtagh 1991], Xgboost [Friedman 2001], and random forest [Breiman 2001]. Once the failure probability of test cases is calculated, these can be ordered considering their failure probabilities.

3.5. Test Case Ordering

The failure probability generated by the classification model for each test case is aggregated with the identifiers of the test cases. In the final stage of the proposed framework, the test cases are ordered to prioritize those with higher failure probabilities. Therefore, the framework returns the sequence of execution identifiers for the test cases.

To verify the performance of our framework, an experimental protocol was designed to replicate a scenario of executing test plans. In these experiments, 5 different versions of the framework were tested using 5 classification techniques. The experimental protocol and the obtained results are presented in the next section.

4. Experiments

In this section, we describe the experimental protocol, evaluation metrics, and results obtained in the experiments. First, however, we present details about the dataset used.

4.1. Dataset

The dataset used in our experiments was provided by a large multinational corporation. The content of the dataset meets the requirements of the proposed scenario and allows us to validate the results of the framework and compare them with those obtained by the strategy adopted by this company. The dataset is originally divided into three sets, each serving a purpose of information representation.

The first is the *test case set*, which has 13,890 instances, containing labeled information regarding the functionalities under test by each individual test case and the actual text of the test script. The second set is the *product set*, which contains codification of 943,469 commit messages from 9052 products, distributed across approximately 240 classes corresponding to the type of component altered in the product. Due to the confidential nature of the commit messages, the company provided a sample of over 50,000 commits, which served to explore techniques for filtering and encoding that could be candidates for execution on the dataset by the company itself. With the filters and codification model defined, the company executed the processing of text from commits and provided a dataset containing the encoded vectors of commits. Finally, the third set is the *execution record set*, which contains information about the application of test cases to real products and the results of their executions. This set consists of 142,401 test case execution records.

4.2. Experimental Protocol

The experiments were conducted using the Hold-Out protocol and taking into account a temporal separation, i.e., by considering the time at which the products available in the dataset had their tests initiated. The proportion used was 70% of the products (necessarily the oldest) for training and the remaining 30% for testing (the most recent ones). The dataset was divided in this way since it is the procedure that most resembles the real process in production. In the training partition, 89.6% of the instances (test case executions) are from the non-failure class, while executions labeled as failure correspond to 1.9% of the set. On the other hand, in the testing partition, instances corresponding to the non-failure class represent 87.3%, while failures correspond to 2.2%. In both partitions, the remaining instances belong to classes such as postponed execution, conditional pass, and

others that regard specific situations of the testing stage and were not considered in this experiment.

Given the two sets obtained after splitting the dataset, the training set and test set, all classification models were submitted to the same experimental protocol. Using the training partition, we manually set the hyperparameters of the models to optimize our classification metrics. After defining the hyperparameters of the models, we executed the final training and evaluated the models on the testing partition.

4.2.1. Comparison Protocol

In order to better evaluate the models' performance considering the ordering task, we employ three additional approaches as baselines: Real Order, Random Order, and Optimal Order of execution. The first considers the sequence used by the testing team of the partner company. Therefore, this method reflects the behavior of human Q&A analysts at the task of ordering the execution of test cases.

The remaining two methods are automatic methods for ordering the execution of test cases. The Random Order is a method that randomly guesses the order of the execution. It was designed by randomly generating 100 sequences for each test plan and presenting the mean result from these sequences. The sequences disregard any information about the product content or the test cases during their generation. Finally, the Optimal Order is an upper bound of ordering strategies, where it always generates the perfect execution sequence for any test plan, consistently achieving the best possible result under the observed metric.

4.3. Metrics

4.3.1. Classification

The metrics evaluated for the classification models are: accuracy, precision, recall, and F1 score. The values presented in the experiments section are the means of the observed test plan values.

4.3.2. Ordering

To evaluate the ordering, we use the APFD (Average Percentage Faults Detected) metric[Pradeepa and VimalDevi 2013], which is usual for test case priorization tasks. This metric measures the ordering of a test case based on the time at which existing faults are found. Its equation is as follows:

$$APFD = 1 - \frac{TF1 + TF2 + ... + TFm}{nm} + \frac{1}{2n} \tag{1}$$

where TFm indicates how many failures were found by the m^{th} executed test, m represents the total number of failures and n represents the number of test cases used.

Similarly to the classification metrics, APFD will also be reported considering the average of the values achieved by the orderings for test plans.

4.4. Results

Initially, we evaluated the five ML models investigated in terms of classification of test cases into classes failure and non-failure. For this purpose, we considered the classification metrics accuracy, precision, recall, and F1-score. As shown in Figure 3, although four metrics have been used, all indicate the following order of decreasing performance: Xgboost, Knn, Logistic Regression, Random Forest, and MLP. However, it is important to note that Logistic Regression, Random Forest, and MLP exhibit very similar results. Xgboost, on the other hand, achieved inferior results.

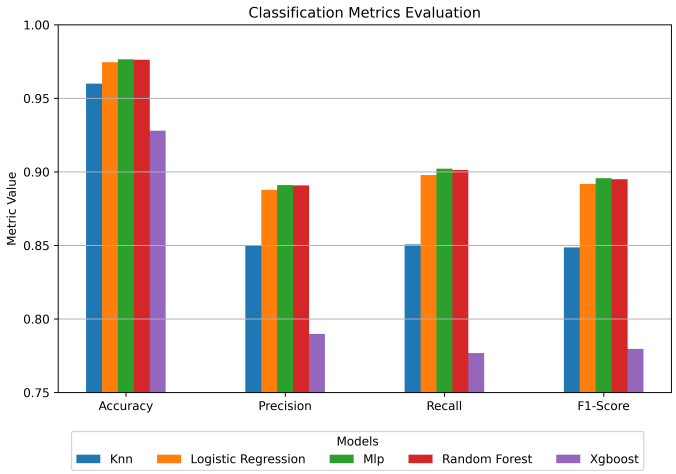

Figura 3. Comparing Machine Learning Algorithms for Classifying Test Cases into Failure and Non-Failure Classes.

As previously mentioned, the objective of our framework is to prioritize the test cases that will result in failures. Therefore, classification is only one necessary step towards ordering. The APFD metric is used to allow us to evaluate the actual ordering results. Here, two APFD values were calculated: one from the real-world data ordering (Real Order) and another from the ordering provided by our framework (Proposed Order). Consequently, for each test plan, we have access to two orderings: Real Order and Proposed Order, whose APFD values were calculated and compared to obtain information on the quality of the orderings for each test plan.

The five classification models were used individually to provide the probability of failure for the ordering step, enabling the testing of five different versions of the framework. Considering these five versions, we observed three possible scenarios for analyzing the ordering quality: (1) Better - scenario where the APFD calculated for the Proposed Order is higher than the APFD calculated for the Real Order; (2) Equal - where both values are similar; and (3) Worse - where the APFD calculated for the Proposed Order is lower than the APFD calculated for the Real Order.

Figure 4 illustrates the behavior of the five framework versions across the three scenarios. For all methods, the test plans are ordered from worse to best APFD results in comparison with its performance in the Real Order. Thus, in the left side are the worst results and on the right side the best results, and the relative improvement (or not) in APFD of the methods can be more easily compared.

Considering the models and scenarios, it is possible to observe the following order of results from worst to best: Knn, Xgboost, Logistic Regression, MLP, with Random Forest obtaining the best results, with 11,79% of the test plans having APFD results above the real order and only 5,76% of them having worse results. It is important to note that, in Figure 4, the width of the areas represents the number of cases where each scenario occurs, while the height of the bars indicates the difference observed in the quality of the ordering.

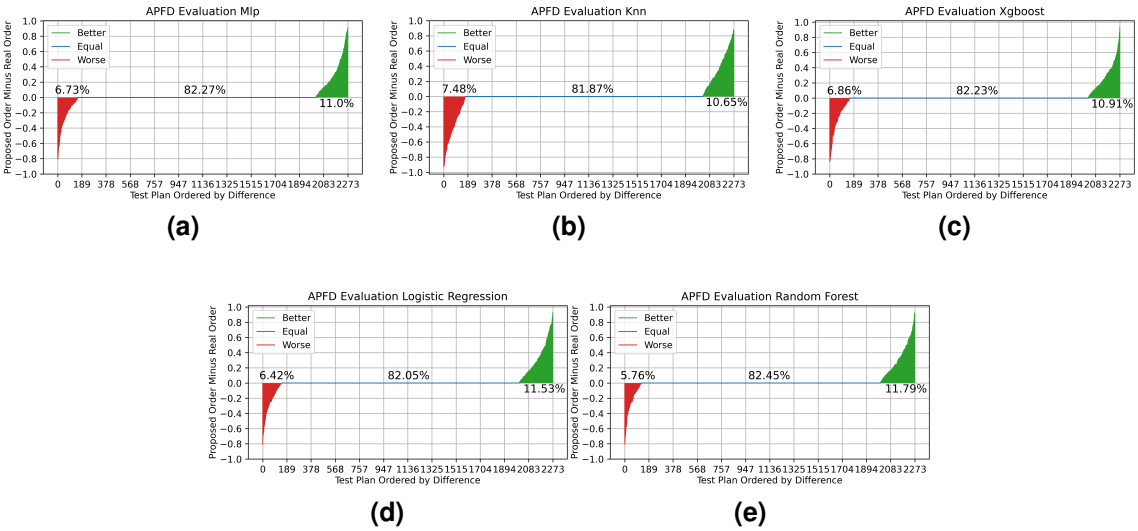

Figura 4. Comparison of the proposed framework using different machine learning algorithms in terms of Actual Percentage of Fault Detection (APFD) against Real APFD. Test plans are ordered from worst to best APFD difference in comparison to to the real order.

To further analyze the impact of our framework in the ordering, Figure 5 presents Violin plots of the APFD differences for test cases with better (a) and worse (b) results when compared to the Real Order, respectively. As can be seen, the worse results of KNN were more frequent and presented larger drops in APFD compared to the other classifiers, which had similar results in that regard. When analyzing the test cases with better APFD, surprisingly KNN had a higher mean APFD difference, but had a smaller number of test cases and highest values compared to all other methods evaluated, which also obtained similar results in comparison with each other, with Random Forest and Logistic Regression having more test cases with higher differences.

4.5. Discussions

Considering the various proposed orderings based on the five versions of the classification models, we can observe that despite the fact that a significant portion of the test plan orderings do not present real gains (Equal), the number of test plans with improved ordering (Better) exceeds the number of worse orderings in all scenarios, as shown in Figure 4.

Therefore, when considering the benefit provided by the framework as the number of plans evaluated as Better minus the number of plans with Worse outcomes, we observe that the framework demonstrates an improvement of 3.17% in its worst-performing classifier version (KNN 10.65% - 7,48%) and 6.03% in its best-performing classifier version

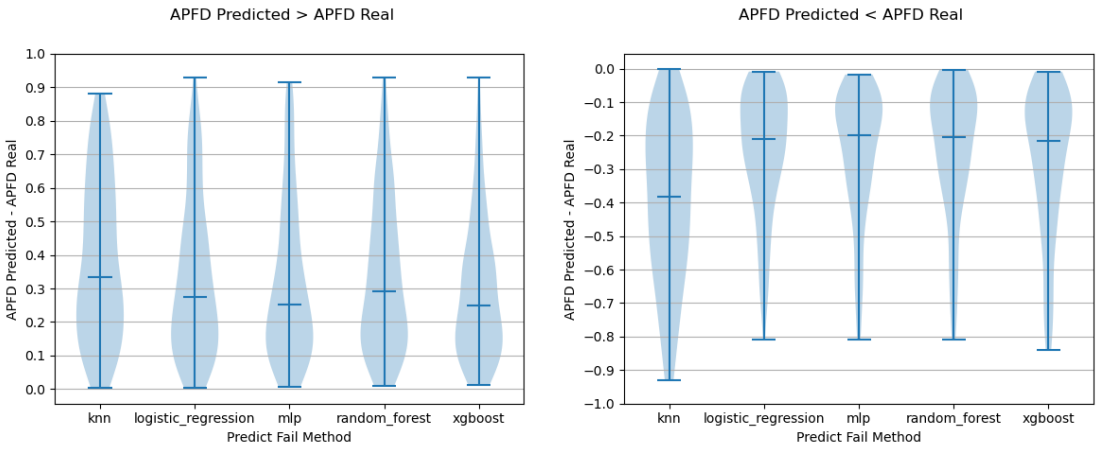

Figura 5. Distribution of differences between predicted and Real APFD for test cases where the framework obtained superior (a) and inferior (b) results in comparison to the Real Order.

(Random Forest 11,79% - 5,76%). Furthermore, our framework attained results superior to or equal to the manual method in between 92.52% and 94.24% of cases, depending on the machine learning algorithm used.

Based on these results, we can affirm that using the test case script and commit logs from a product submitted to this testing succeeded in providing sufficient information to the classification models to achieve a gain of up to 6.03% in the ordering of test plans, using the probability of failure as the ordering criterion.

Furthermore, as can be seen in Figure 4, there is a large number of test plans with APFD value tied between the real order and the order given by our framework. Most of these ties can be explained by the fact that 82.08% of test plans executed in the test set there were no test cases that ended in failure, hence having an APFD value of 1 irregardless of the test case ordering. Due to these cases being the majority, eventual improvements made from our framework may get diluted when analyzing ordering metrics.

Thus, to better visualize the impact of our Framework in the test plans where it *can* make any difference, Figure 6 presents the mean APFD for the different classification methods of our Framework, as well as the Real, Random and Optimal Order APFD results, considering only test plan executions where at least one failure was found. In these cases, the proposed ranking strategy once again demonstrated superior performance, achieving improvements of nearly 10% over the manually performed ordering. However, despite the gains obtained compared to the Real Order and Random Order, there remains a significant gap when contrasted with the Optimal Order. Even in the best scenario (Random Forest), the difference still stands at approximately 30% indicating that there is still room for improvements in the framework.

5. Conclusions

In this work, we propose a framework for ordering regression test plans considering non-automated and black-box testing cases. The ordering is performed based on the probability of classification of a test case execution on a specific product as a failure, which is

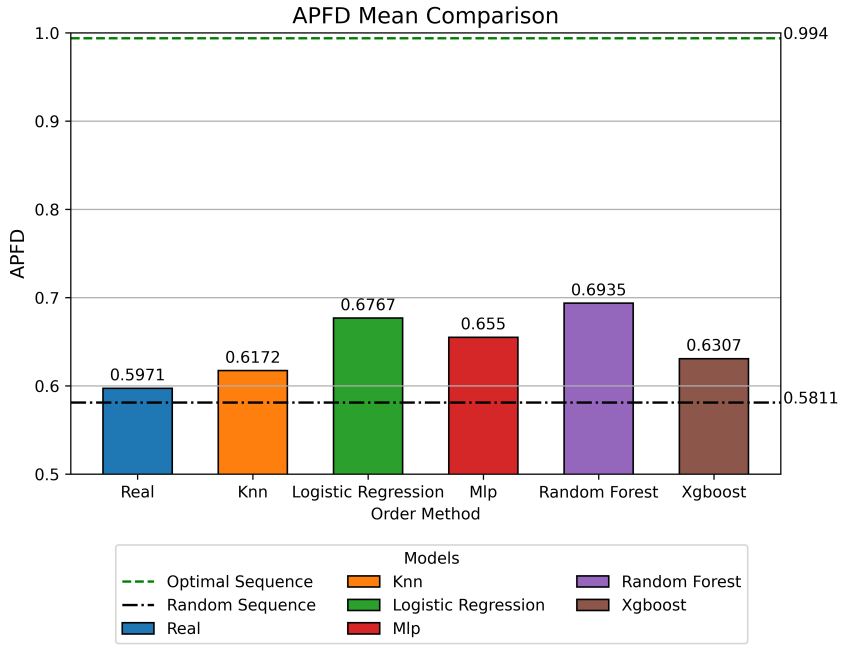

Figura 6. Proposed framework compared to Real, Random and Optimal Order.

calculated by machine learning models. The information used to represent each test case includes textual natural language information from the test case and a product change log to associate this pair with a failure or not.

We evaluated five different versions of the framework by varying the machine learning method used to classify the test cases. The results of the framework were compared to the performance of the sorting made by an expert team from the company owning the data, as well as random and optimal orderings of test case execution. The experiment results showed that the adopted ordering strategy achieved positive gain-loss ratios in all scenarios. Although there was similar performance between automatic and manual ordering in many cases, this scenario only reinforces the good performance of the framework, as our solution was able to equal or outperform the expert team with previously acquired experience in this function.

Another noteworthy point is that the information from commits and test case scripts proved capable of providing indications of a probable failure in the tested product, given that no source code was used.

The threats to the validity of this experiment reside in its use of proprietary data, making it difficult to generalize its performance to other data scenarios. Another point is the impossibility of validating the obtained results by disregarding the intrinsic characteristics of the test set, due to the necessary Hold-Out design feature of the problem.

Finally, new approaches are still needed for the investigated problem, such as expanding the framework's information sources and testing new machine learning model approaches like neural networks with higher learning capacity and techniques like reinforcement learning, where it would also be worthwhile to explore the use of the APFD metric as a reward for the model.

Artifact Availability

Due to our partner industrial compliance, we cannot provide any software developed during this research. Some parts contain sensitive content attached or industrial secrets that can be retrieved into our framework source code. Hence, since this research was conducted on texts extracted from commits and tests scripts from real data, it contains employee information or secrets about products functions. Consequently, even coded, the data and trained models are not allowed to being provided. Moreover, simulated versions of sensitive functions or content would turn the data useless and compromise results, as artificial generated data could not reproduce real-world data. Considering this context, we are unable to elaborate software or data artifacts that can be followed, comprehended, or executed to really reproduce results without compromising the expected confidentially.

Acknowledgments

This research was partially supported by Motorola Mobility Comércio de Produtos Eletrônicos Ltda, under the terms of Federal Law No. 8.387/1991, through agreement No. 02/2021, signed with ICOMP/UFAM. Also, this work was partially supported by the following Brazilian agencies: the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior- Brasil (CAPES-PROEX)- Finance Code 001, the National Council for Scientific and Technological Development (CNPq), and Amazonas State Research Support Foundation- FAPEAM- through the POSGRAD project 2024/2025.

Referências

Al-Sabbagh, K., Staron, M., Hebig, R., and Gomes, F. (2021). A classification of code changes and test types dependencies for improving machine learning based test selection. In *Proceedings of the 17th International Conference on Predictive Models and Data Analytics in Software Engineering*, pages 40–49.

An, G. and Yoo, S. (2022). Fdg: a precise measurement of fault diagnosability gain of test cases. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 14–26.

Batista, G., Silva, D. F., et al. (2009). How k-nearest neighbor parameters affect its performance. In *Argentine symposium on artificial intelligence*, pages 1–12. Citeseer.

Breiman, L. (2001). Random forests. *Machine learning*, 45:5–32.

Brzezinski, J. R. and Knafl, G. J. (1999). Logistic regression modeling for context-based classification. In *Proceedings. Tenth International Workshop on Database and Expert Systems Applications. DEXA 99*, pages 755–759. IEEE.

Friedman, J. H. (2001). Greedy function approximation: a gradient boosting machine. *Annals of statistics*, pages 1189–1232.

Mafra, J., Miranda, B., Iyoda, J., and Sampaio, A. (2009). Test case selector: Uma ferramenta para seleção de testes. *Proceedings of SBMF/SAST*.

Mehta, S., Farmahinifarahani, F., Bhagwan, R., Guptha, S., Jafari, S., Kumar, R., Saini, V., and Santhiar, A. (2021). Data-driven test selection at scale. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1225–1235.

Murtagh, F. (1991). Multilayer perceptrons for classification and regression. *Neurocomputing*, 2(5-6):183–197.

Omri, S. and Sinz, C. (2022). Learning to rank for test case prioritization. In *Proceedings of the 15th Workshop on Search-Based Software Testing*, pages 16–24.

Palma, F., Abdou, T., Bener, A., Maidens, J., and Liu, S. (2018). An improvement to test case failure prediction in the context of test case prioritization. In *Proceedings of the 14th international conference on predictive models and data analytics in software engineering*, pages 80–89.

Pan, C., Yang, Y., Li, Z., and Guo, J. (2020). Dynamic time window based reward for reinforcement learning in continuous integration testing. In *Proceedings of the 12th Asia-Pacific Symposium on Internetware*, pages 189–198.

Pradeepa, R. and VimalDevi, K. (2013). Effectiveness of testcase prioritization using apfd metric: Survey. In *International Conference on Research Trends in Computer Technologies (ICRTCT—2013). Proceedings published in International Journal of Computer Applications®(IJCA)*, pages 0975–8887.

Ramírez, A., Feldt, R., and Romero, J. R. (2023). A taxonomy of information attributes for test case prioritisation: Applicability, machine learning. *ACM Transactions on Software Engineering and Methodology*, 32(1):1–42.

Reimers, N. (2019). Sentence-bert: Sentence embeddings using siamese bert-networks. *arXiv preprint arXiv:1908.10084*.

Rizwan, S., Ali Sobuj, M. S., and Akhond, M. R. (2022). A survey on software test case minimization. In *Proceedings of the 2022 Fourteenth International Conference on Contemporary Computing*, pages 679–684.

Shankar, R. and Sridhar, D. (2024). An improved deep learning based test case prioritization using deep reinforcement learning. *International Journal of Intelligent Engineering & Systems*, 17(1).

Spieker, H., Gotlieb, A., Marijan, D., and Mossige, M. (2017). 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*, pages 12–22.

Tahvili, S., Hatvani, L., Felderer, M., Afzal, W., Saadatmand, M., and Bohlin, M. (2018). Cluster-based test scheduling strategies using semantic relationships between test specifications. In *Proceedings of the 5th International Workshop on Requirements Engineering and Testing*, pages 1–4.

Wu, Z., Yang, Y., Li, Z., and Zhao, R. (2019). A time window based reinforcement learning reward for test case prioritization in continuous integration. In *Proceedings of the 11th Asia-Pacific Symposium on Internetware*, pages 1–6.

Zhang, J., Liu, Y., Gligoric, M., Legunsen, O., and Shi, A. (2022). Comparing and combining analysis-based and learning-based regression test selection. In *Proceedings of the 3rd ACM/IEEE International Conference on Automation of Software Test*, pages 17–28.